CS2800 Fall 2010 - Lecture 23

Harsh Raju Chamarthi

10 November 2010

1 Inductive proofs of implication formulas

There was a quiz, which was:
Q1: What is the induction scheme generated by `mul`[1].
Q2: What are the proof obligations we get when we apply the above induction scheme to `(= (mul n m) (* n m))`.

In the last class we proved that subset is reflexive(i.e any set is a subset of itself: (subset A A)). We used a lemma to prove it, but that proof will fall apart, if we dont prove the lemma. So lets prove the lemma:

```
[subset-cons-element]
(implies (subset A B)
         (subset A (cons x B)))
```

We dont have any basic lemmas about subset, so we have no choice but to carry out a proof by induction, which means that to prove the above lemma, I need to apply an induction scheme, which scheme should I choose, I will choose (tlp A) or (subset A B), but both have the same induction schemes(Note that the reason why (subset A B) terminates is the same reason why (tlp A) terminates).

So if I apply the induction scheme of (tlp A) to the above formula, I have the following **proof obligations**: **base case:**

[1] See Lecture 20

```
(implies (endp A)
         (implies (subset A B)
                  (subset A (cons x B))))
```

which can be clearly simplified using the propositional tautology:

$$p \to q \to r \equiv p \wedge q \to r$$

to

```
(implies (and (endp A)
              (subset A B))
         (subset A (cons x B))))
```

If we prove this simplified formula, we know by the Rule of inference of Prop. Deduction, we can deduce our original formula.

Lets write down the formula we are going to reduce to `t` and the context:

```
(subset A (cons x B))
⇐ { def. subset, BCcond }
  t
```

Context
BCcond:`(endp A)`
A1: `(subset A B)`

The other proof obligation is the
induction step:

```
(implies (and (not (endp A))
              (implies (subset (cdr A) B)
                       (subset (cdr A) (cons x B))))
         (implies (subset A B)
                  (subset A (cons x B))))
```

which can again be simplified using our standard technique of pushing all the antecedents in the conclusion into the top-level implication:

```
(implies (and (not (endp A))
              (implies (subset (cdr A) B)
                       (subset (cdr A) (cons x B)))
              (subset A B))
         (subset A (cons x B)))
```

Which means all we need to prove is `(subset A (cons x B))`, assuming all the formulas in the antecedent(these formulas are also called hypotheses). Note, in the context I have abbreviated `subset` to `s`.

```
(subset A (cons x B))
⇐ {def. subset, IS1}
(and (in (car A) (cons x B))
     (subset (cdr A) (cons x B)))
⇐ {def in, consp-cons, car-cdr-cons}
(and (or (= (car A) x)
         (in (car A) B))
     (subset (cdr A) (cons x B)))
⇐ {Prop. Deduction, D1 }
(subset (cdr A) (cons x B))
⇐ {D3 }
t
```

Context

```
IS1: (not (endp A))
IH:
(implies (s (cdr A) B)
         (s (cdr A) (cons x B)))
A1: (subset A B)

by {def. subset, IS1}
we can deduce:
D1: (in (car A) B)
D2: (s (cdr A) B)
from {Prop. Ded, D2, IH }
we can furthur deduce:
D3: (s (cdr A) (cons xB))
```

Beware: You can only push the antecedents in the conclusion of the top-level implication to the top, **not** the antecedents of the implication in the antecedent of the top-level implication.

Notes:

1. When starting a proof, ask yourself the question, does this proof even need induction. Maybe there are some lemmas i can use, if not then you decide, okay, this property is about recursive functions, so I have no choice but to **try** an induction proof.

2. Once you have decided on applying induction, the next question you must answer is "What Induction Scheme should I use?", the answer of which is *Choose the most important function in the formula you want to prove*. But that really does not say much. Remember if the property you are trying to prove is inherently about true-lists, you can get away using the induction scheme of `(tlp X)` most of the time. But this is not always the case, as we will see shortly.

3. Repeat again: "I cannot instantiate any formula in my context". Assumptions can never be instantiated, since they are not true statements, they are true only for the current proof. Only Axioms and

theorems(which are true statements for all acl2 objects) can be instantiated.

4. The most powerful assumption in the context, in an induction proof, is the Induction Hypothesis(IH), if ever in doubt, and you have a choice, you must choose to proceed the proof in a manner which would take you as close as possible to a situation where you can **use** the Induction Hypothesis.

5. Extending the initial context to get **derived** context is a a very nice space-saving technique and intuitive. You dont use this technique all the time, but its especially used in induction proofs of formulas of the form `(implies A C)`. See next point.

6. In the previous proof(after applying the first 2 reasons), I need to know something about `(subset (cdr A) (cons x B))`, in fact I need to show its true, but all I know from the context is that `(implies (subset (cdr A) B) (subset (cdr A) (cons x B)))` is true, which means, I cant use it right away. You can think of it, as being locked, you need to use whats in the conclusion of the IH, but it cant be used unless you unlock it, i.e show that the antecedent of the IH is true. The good news is that this can always be done:
   From the rest of the (initial) context, i.e IS1 and A1, I can easily deduce `(subset (cdr A) B)`, which unlocks my induction hypothesis by the propositional tautology(Modus Ponens):

   $$p \wedge (p \rightarrow q) \rightarrow q$$

   Thus using the rest of the initial context, you can always deduce the conclusion of the induction hypothesis implication, which can be directly used in the proof. This technique is repeated for all induction proofs of implication formulas(i.e `(implies A C)`).

2 Proving correctness of accumulator-style functions

Now we will see how to prove that an efficient version of a function written in accumulator-passing style(tail-recursive) gives the same answers as the original version.

Here are definitions:

```
;; rev : tlp -> tlp
(defun rev (x)
  "Reverse a list"
  (if (endp x)
      nil
    (app (rev (cdr x))
         (list (car x)))))

;; rev*-acc : tlp tlp -> tlp
(defun rev*-acc (x acc)
 "cons elements of x onto acc, thus reversing x"
  (if (endp x)
    acc
    (rev*-acc (cdr x)
              (cons (car x) acc))))

;; rev*: tlp -> tlp
(defun rev* (x)
  "Tail-recursive version of reverse"
  (rev*-acc x nil))
```

I want to prove that `rev*` is correct. What does it mean? Well if we know `rev` works correctly, all I need to show is:

```
(= (rev* x) (rev x))
```

But wait, rev* is a non-recursive function, normally I would like to use a lemma to prove this(without using induction), but unfortunately I dont have any lemmas that would help, but in the process of proving the above you will notice the process of how to come up with the lemma. As I said `rev*` just expands to `(rev*-acc x nil)`. Which means we need to prove `(= (rev*-acc x nil) (rev x))`. Since this is a property about recursive functions(which recur on true-lists), I need to use Induction, as I currently have no lemmas to help me. At this point you should ask, what Induction scheme should I use. There are 3 choices, `(rev x)`, `(tlp x)` or `(rev*-acc x acc)`, but the first two are exactly the same induction schemes(why? simple, on a piece of paper write down the induction schemes for both functions and see for yourself).

Lets choose the induction scheme given by (rev x), then we get the following

proof obligations:

```
Base Case:
(implies (endp x)
         (= (rev*-acc x nil) (rev x)))
Induction Step:
(implies (and (not (endp x))
              (= (rev*-acc (cdr x) nil) (rev (cdr x))))
         (= (rev*-acc x nil) (rev x)))
```

Base case is simple, lets do the IS:

Context

```
IS1: (not (endp x))
IH: (= (rev*-acc (cdr x) nil)
       (rev (cdr x)))
```

```
(= (rev*-acc x nil)
   (rev x))
⇐ {def. rev*-acc,rev, IS1}
(= (rev*-acc (cdr x)
             (cons (car x) nil))
   (app (rev (cdr x))
        (list (car x))))
⇐ { ?????  }
```

I want to use my Induction Hypothesis(IH), but I cannot, and no amount of opening definitions of rev/rev*-acc will help me get there, since the second argument always increases. If I cannot use the IH, there is no point in doing an induction proof anymore. This might be an instance of trying to prove something using a wrong induction scheme, which is not rare, but as we will shortly see this is an instance of trying to use induction to prove a theorem which is not general enough. To apply induction in its full power and glory, one needs to generalize[2]. But more on that later, lets see another failed proof:

Lets induct on (rev*-acc x acc). What induction scheme does it have, lets write it down:

[2]This generalization falls right out of this failed proof

Induction scheme of `(rev*-acc x acc)`:

```
Base Case:
(implies (endp x)  φ)

Induction Step:
(implies (and (not (endp x))
              φ|σ:((x (cdr x)) (acc (cons (car x) acc)))
         φ)
```

If I apply this induction scheme to the formula we are trying to prove, we obtain the following proof obligations:

```
Base Case:
(implies (endp x)
         (= (rev*-acc x nil) (rev x)))
Induction Step:
(implies (and (not (endp x))
              (= (rev*-acc (cdr x) nil) (rev (cdr x))))
         (= (rev*-acc x nil) (rev x)))
```

But wait, this is exactly the same proof obligation as before, so we know we will get stuck again. Why is induction not working? Induction schemes are generated from function definitions, the scheme has `acc`, but `rev` does not have any mention of `acc`, ideally we would like to apply induction on a more general, a stronger formula. What exactly are we trying to accomplish here, well, we want to relate the result of computing `rev*-acc` and `rev`. Since `rev*-acc` has an extra argument, we must take care of it too, for which we have to find the missing function(`f`) in the following relation:

```
(= (rev*-acc x acc)
   (f acc (rev x) x))
or think of it as a fill in the blanks:
(= (rev*-acc x acc)
   (... acc (rev x) ...))
or if you prefer this order:
(= (rev*-acc x acc)
   (... (rev x) acc ...))
```

As Cosimo, correctly guessed, the answer is to combine both the lists. Think about it. `rev*-acc` just walks down the list consing its elements onto an initial accumulator argument. Assuming (len x) = n and (len acc) = m, at the end of the recursion, when we return the answer, it has n+m elements, the first n elements being exactly the reversed list and the remaining m elements in the back are the original accumulator you started with, the relation is therefore simply:

```
Lemma rev-app
(= (rev*-acc x acc)
   (app (rev x) acc))
```

In which case our `f` in the fill in the blanks is the following mathematical function: `(f a1 a2 a3) = (app a2 a1)`.

By the way, if you look at last footnote, I mention that this generalization could easily be guessed from our first failed proof. Now that we have a general lemma(`rev-app`), we can officially call it a lemma only if we prove it, lets prove it, but remember the choice of induction scheme is very important here, since we have `acc` in the formula, it would be beneficial if we choose the induction scheme generated by `rev*-acc`, since it seems to be the most important function in the formula. If we use that induction scheme we have as usual the following proof obligations:

```
Base Case:
(implies (endp x)
         (= (rev*-acc x acc)
            (app (rev x) acc))
Induction Step:
(implies (and (not (endp x))
              (= (rev*-acc (cdr x) (cons (car x) acc))
                 (app (rev (cdr x)) (cons (car x) acc))))
         (= (rev*-acc x acc)
            (app (rev x) acc)))
```

Base case is simple, do it yourself, lets do the Induction step(IS):

Context

```
IS1: (not (endp x))
IH: (= (rev*-acc (cdr x)
                 (cons (car x) acc))
       (app (rev (cdr x))
            (cons (car x) acc)))
```

```
(= (rev*-acc x acc)
   (app (rev x) acc)
⇐ {def. rev*-acc,rev, IS1}
(= (rev*-acc (cdr x)
             (cons (car x) acc))
   (app (app (rev (cdr x)) (list (car x)))
        acc))
⇐ {IH, def. list}
(= (app (rev (cdr x)) (cons (car x) acc))
   (app (app (rev (cdr x)) (cons (car x) nil))
        acc))
⇐ {app-associative lemma}
(= (app (rev (cdr x)) (cons (car x) acc))
   (app (rev (cdr x)) (app (cons (car x) nil)
                           acc)))
⇐ {app-cons lemma}
(= (app (rev (cdr x)) (cons (car x) acc))
   (app (rev (cdr x)) (cons (car x) (app nil acc))))
⇐ {def. app, if-true}
(= (app (rev (cdr x)) (cons (car x) acc))
   (app (rec (cdr x)) (cons (car x) acc)))
⇐ { Equality }
 t
```

If you recall we proved the following two lemmas in disguise in previous lectures.

```
(equal (app (cons a b) c)
       (cons a (app b c)))  app-cons (Lec 16)
```

```
(equal (app (app x y) z)
       (app x (app y z)))  app-associative (Lec 17)

(equal (app x nil) x)    app-nil (Lec 19)
```

Note: In particular if you want to prove anything about a function, you need to prove lemmas about its constituent functions, for e.g since body of `rev` contains `app, cons`, it would help to have lemmas which characterize the properties of app, cons and combinations.

But wait we havent proved our original conjecture, which was `(rev* x) = (rev x)?`. Lets do the proof, but notice we dont have to use any induction, we will use purely the reasoning we did before we learned any induction:

```
(= (rev* x)
   (rev x))
⇐ {def. rev*}
(= (rev*-acc x nil)
   (rev x))
⇐ { Instantiate lemma rev-app}
(= (app (rev x) nil)
   (rev x))
⇐ {instantiate lemma app-nil}
(= (rev x)
   (rev x))
⇐ {equality}
t
```

With that we finish our proof of the fact that the efficient implementation of reverse works correctly(works like our original reverse). All proofs of correctness of accumulator style(tail-recursive) functions will follow the same proof pattern as shown above.