Logic and Computation – CS 2800 Fall 2019

Lecture 30

Accumulators

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

What is the answer to this quiz from last time?

- A function `mysort` which satisfies the claim below is a correct sorting function: YES/NO

```
(defthm mysort-ordered
  (implies (lorp L)
           (orderedp (mysort L))))
```

Outline

- Tail recursion
- Accumulators
- Reasoning about tail-recursive functions

Tail recursion

Consider these functions

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)
      y
    (cons (car x) (aapp (cdr x) y))))

(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))
```

- How many conses does rrev require?
- $O(n^2)$, where n is length of list x.
- Can we do better?

Tail-recursive version

```
(definec revt (x :tl acc :tl) :tl
  (if (endp x)
    acc
    (revt (rest x) (cons (first x) acc))))

(definec rev* (x :tl) :tl
  (revt x nil))
```

- What is “acc”?
- Accumulator

- Is this definition “better” (more efficient) than the previous one?
- Is it semantically equivalent?

Tail-recursive version is better

```
(definec revt (x :tl acc :tl) :tl
  (if (endp x)
    acc
    (revt (rest x) (cons (first x) acc))))

(definec rev* (x :tl) :tl
  (revt x nil))
```

- How many conses does rev* require?
- $O(n)$, where n is length of list x.

Is the new version equivalent to the original one?

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)
      y
    (cons (car x) (aapp (cdr x) y))))

(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))

(definec revt (x :tl acc :tl) :tl
  (if (endp x)  acc
    (revt (rest x) (cons (first x) acc))))

(definec rev* (x :tl) :tl
  (revt x nil))
```

Let's try to prove A1 by induction on true list x.

Claim A0:

(tlp x) => (rev* x) = (rrev x)

Or equivalently, Claim A1:

(tlp x) => (revt x nil) = (rrev x)

Claim A1: induction step

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)  y
    (cons (car x) (aapp (cdr x) y))))
(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))
(definec revt (x :tl acc :tl) :tl
  (if (endp x)  acc
    (revt (rest x) (cons (first x) acc))))
```

```
C1. (tlp x)     C2. (not (endp x))
C3. (tlp (cdr x)) => (revt (cdr x) nil) = (rrev (cdr x))
D1. (tlp (cdr x)) { C1, C2 }
D2. (revt (cdr x) nil) = (rrev (cdr x)) { C3, D1, MP }

Goal: (revt x nil) = (rrev x)

Proof:
(revt x nil) = { def revt, C2 }
(revt (cdr x) (cons (car x) nil)) = ???
```

Need generalization!
But what?

Cannot use D2!

Claim A2: generalized

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)  y
    (cons (car x) (aapp (cdr x) y))))
(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))
(definec revt (x :tl acc :tl) :tl
  (if (endp x)  acc
    (revt (rest x) (cons (first x) acc))))
```

```
Claim A2 (generalization) should look like:

(tlp x) & (tlp acc)
     => (revt x acc) = ???
```

Claim A2: generalized

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)  y
    (cons (car x) (aapp (cdr x) y))))
(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))
(definec revt (x :tl acc :tl) :tl
  (if (endp x)  acc
    (revt (rest x) (cons (first x) acc))))
```

```
Claim A2 (generalization):

(tlp x) & (tlp acc)
     => (revt x acc) = (aapp (rrev x) acc)
```

Does the generalized claim A2 help us prove the original claim A1?

```
Claim A1 (original):
(tlp x) => (revt x nil) = (rrev x)
```

Claim A2: attempt to prove by induction on true list x

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)  y
    (cons (car x) (aapp (cdr x) y))))
(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))
(definec revt (x :tl acc :tl) :tl
  (if (endp x)  acc
    (revt (rest x) (cons (first x) acc))))
```

```
C1. (tlp x)  C2. (tlp acc)  C3. (not (endp x))
...
Dn. (revt (cdr x) acc) = (aapp (rrev (cdr x)) acc)

Goal:       (revt x acc) = (aapp (rrev x) acc)
Proof:
(revt x acc) = { def revt, C3 }
(revt (cdr x) (cons (car x) acc)) = ???
```

Again we cannot use the induction hypothesis Dn!

Remember: we cannot instantiate context!!!

Claim A2: proof by induction on ???

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)  y
    (cons (car x) (aapp (cdr x) y))))
(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))
(definec revt (x :tl acc :tl) :tl
  (if (endp x)  acc
    (revt (cdr x) (cons (car x) acc))))
```

```
C1. (tlp x)   C2. (tlp acc)   C3. (not (endp x))
...
Dn. (revt (cdr x) acc) = (aapp (rrev (cdr x)) acc)

Goal:        (revt x acc) = (aapp (rrev x) acc)
Proof:
(revt x acc) = { def revt, C3 }
(revt (cdr x) (cons (car x) acc)) = ???
```

We need another induction hypothesis => another induction scheme. But which one? The one of revt!

Claim A2: proof by induction on (revt x acc)

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)  y
    (cons (car x) (aapp (cdr x) y))))
(definec rrev (x :tl) :tl
  (if (endp x)
    nil
    (aapp (rrev (rest x)) (list (first x)))))
(definec revt (x :tl acc :tl) :tl
  (if (endp x)  acc
    (revt (cdr x) (cons (car x) acc))))
```

```
C1. (tlp x)  C2. (tlp acc)  C3. (not (endp x))   ...
Dn. (revt (cdr x) (cons (car x) acc))
    = (aapp (rrev (cdr x)) (cons (car x) acc))

Goal:          (revt x acc) = (aapp (rrev x) acc)
Proof:
(revt x acc) = { def revt, C3 }
(revt (cdr x) (cons (car x) acc)) = { Dn }
(aapp (rrev (cdr x)) (cons (car x) acc)) = { def aapp }
(aapp (rrev (cdr x)) (aapp (list (car x)) acc)) = {aapp assoc. }
(aapp (aapp (rrev (cdr x)) (list (car x))) acc) = {def rrev,C3 }
(aapp (rrev x) acc)
```

Reasoning about tail-recursive functions

The recipe – part 1: defining tail-recursive functions

1. Start with the original function `f`
2. Define a tail-recursive version `ft` of `f`, with an accumulator `acc`
3. Define `f*`, a non-recursive version that calls `ft` with appropriate arguments and is logically equivalent to `f`, i.e., the following theorem holds:

```
Hypotheses => (f* ...) = (f ...)
```

The recipe – part 2: reasoning about tail-recursive functions

4. Identify a lemma that relates `ft` to `f` . This lemma should have the form:

```
Hypotheses => (ft ... acc) = ... (f ...) ...
```

Remember that you must generalize, so all arguments to `ft` must be variables, not constants. The RHS should include `acc`.

5. Assuming the lemma above is true (don't try to prove it yet!) and using only equational reasoning, and maybe some new lemmas, prove the main theorem:

```
Hypotheses => (f* ...) = (f ...)
```

6. Prove the lemma in step 4 using the induction scheme of `ft`.
7. Prove any remaining lemmas.

Next time

- Induction continued