Logic and Computation – CS 2800 Fall 2019

Lecture 31

Accumulators continued

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- Reasoning about tail-recursive functions
- More examples

Example 1: original version

```
(definec pow (x :rational p :nat) :rational
  (if (equal p 0)
      1
    (* x (pow x (- p 1)))))
```

- Is it tail-recursive?
- What would tail-recursive version be?

Example 1: tail-recursive version

```
(definec pow (x :rational p :nat) :rational
  (if (equal p 0)
      1
    (* x (pow x (- p 1)))))

(definec pow-t (x :rational p :nat acc :rational) :rational
  (if (equal p 0)
      acc
    (pow-t x (- p 1) (* x acc))))

(definec pow* (x :rational p :nat) :rational
  (pow-t x p 1))
```

- What would the lemma and main theorem of the recipe be?

Example 1: lemma and theorem

```
(definec pow (x :rational p :nat) :rational
  (if (equal p 0)
      1
    (* x (pow x (- p 1)))))

(definec pow-t (x :rational p :nat acc :rational) :rational
  (if (equal p 0)
      acc
    (pow-t x (- p 1) (* x acc))))

(definec pow* (x :rational p :nat) :rational
  (pow-t x p 1))

(defthm pow-t-lemma
  (implies (and (rationalp x) (natp p) (rationalp acc))
           (equal (pow-t x p acc) (* acc (pow x p)))))

(defthm pow*-thm
  (implies (and (rationalp x) (natp p))
           (equal (pow* x p) (pow  x p))))
```

Prove these at home!

Use the recipe!

Example 2: replace-all

```
(definec replace-all (e :all f :all l :tl) :tl
  (cond ((endp l) l)
        ((equal e (first l))
         (cons f (replace-all e f (rest l))))
        (t (cons (first l) (replace-all e f (rest l))))))


(replace-all '(1) 2 '(1 (2) (1) 2 (1) 1))
```

- Is it tail-recursive?
- What would tail-recursive version be?

Example 2: replace-all, tail-recursive version: rt1 and r*1

```
(definec replace-all (e :all f :all l :tl) :tl
  (cond ((endp l) l)
        ((equal e (first l))
         (cons f (replace-all e f (rest l))))
        (t (cons (first l) (replace-all e f (rest l))))))

;; tail-recursive version of replace-all, version 1:
(definec rt1 (e :all f :all l :tl acc :tl) :tl
  (cond ((endp l) (rev* acc))
        ((equal e (first l))
         (rt1 e f (rest l) (cons f acc)))
        (t (rt1 e f (rest l) (cons (first l) acc)))))

(definec r*1 (e :all f :all l :tl) :tl
  (rt1 e f l nil))
```

- Lemma and main theorem of the recipe?

Example 2: replace-all, test, lemma and theorem for rt1 and r*1

```
(test?
 (implies (tlp l)
          (equal (r*1 e f l)
                 (replace-all e f l))))

(defthm rt1-lemma
  (implies (and (tlp l) (tlp acc))
           (equal (rt1 e f l acc)
                  (aapp (rrev acc) (replace-all e f l)))))

(defthm r*1-thm
  (implies (tlp l)
           (equal (r*1 e f l)
                  (replace-all e f l))))
```

Prove these at home!

Use the recipe!

Example 2: replace-all, tail-recursive version: rt2 and r*2

```
(definec replace-all (e :all f :all l :tl) :tl
  (cond ((endp l) l)
        ((equal e (first l))
         (cons f (replace-all e f (rest l))))
        (t (cons (first l) (replace-all e f (rest l))))))

;; tail-recursive version of replace-all, version 2:
(definec rt2 (e :all f :all l :tl acc :tl) :tl
  (cond ((endp l) acc)
        ((equal e (first l))
         (rt2 e f (rest l) (cons f acc)))
        (t (rt2 e f (rest l) (cons (first l) acc)))))

(definec r*2 (e :all f :all l :tl) :tl
  (rt2 e f (rev* l) nil))
```

- Lemma and main theorem of the recipe?

Example 2: replace-all, test, lemma and theorem for rt2 and r*2

```
(test?
 (implies (tlp l)
          (equal (r*2 e f l)
                 (replace-all e f l))))

(defthm rt2-lemma
  (implies (and (tlp l) (tlp acc))
           (equal (rt2 e f l acc)
                  (aapp (rrev (replace-all e f l)) acc))))

(defthm r*2-thm
  (implies (tlp l)
           (equal (r*2 e f l)
                  (replace-all e f l))))
```

Prove these at home!

Use the recipe!

Example 2: replace-all, tail-recursive version: r*3

```
(definec replace-all (e :all f :all l :tl) :tl
  (cond ((endp l) l)
        ((equal e (first l))
         (cons f (replace-all e f (rest l))))
        (t (cons (first l) (replace-all e f (rest l))))))

;; tail-recursive version of replace-all, version 2:
(definec rt2 (e :all f :all l :tl acc :tl) :tl
  (cond ((endp l) acc)
        ((equal e (first l))
         (rt2 e f (rest l) (cons f acc)))
        (t (rt2 e f (rest l) (cons (first l) acc)))))

;; tail-recursive version of replace-all, version 3 (uses rt2):
(definec r*3 (e :all f :all l :tl) :tl
  (rev* (rt2 e f l nil)))
```

- Lemma and main theorem of the recipe?

Example 2: replace-all, test and theorem for r*3 (the lemma will still be the one for rt2)

```
(test?
 (implies (tlp l)
          (equal (r*3 e f l)
                 (replace-all e f l))))

(defthm r*3-thm
  (implies (tlp l)
           (equal (r*3 e f l)
                  (replace-all e f l))))
```

Prove this at home!
Use the recipe!

Example 3: add-lists

```
(defdata lor (listof rational))

(definec add-lists (x :lor y :lor) :lor
  (cond ((endp x) y)
        ((endp y) x)
        (t (cons (+ (car x) (car y))
                 (add-lists (cdr x) (cdr y))))))

(add-lists '(1 2 3) '(4 5))
```

- Is it tail-recursive?
- What would tail-recursive version be?

Example 3: add-lists, tail-recursive version

```
(definec add-lists-t (x :lor y :lor acc :lor) :lor
  (cond ((endp x) (aapp (rev* acc) y))
        ((endp y) (aapp (rev* acc) x))
        (t (add-lists-t (cdr x)
                        (cdr y)
                        (cons (+ (car x) (car y)) acc)))))

;; Notice the use of rev* above: we want execution to be fast.

(definec add-lists* (x :lor y :lor) :lor
  (add-lists-t x y nil))
```

- Lemma and main theorem of the recipe?

Example 3: add-lists, test, lemma and theorem

```
(test? (implies (and (lorp x) (lorp y))
                (equal (add-lists* x y)
                       (add-lists x y))))

(test?
 (implies (and (lorp x) (lorp y) (lorp acc))
          (equal (add-lists-t x y acc)
                 (aapp (rrev acc) (add-lists x y)))))

(defthm add-lists-t-lemma
  (implies (and (lorp x) (lorp y) (lorp acc))
           (equal (add-lists-t x y acc)
                  (aapp (rrev acc) (add-lists x y)))))

(defthm add-lists*-thm
  (implies (and (lorp x) (lorp y))
           (equal (add-lists* x y)    (add-lists x y))))
```

Prove these at home!

Use the recipe!

Next time

- Induction continued