CS 457/557 Functional Programming

Lecture 11

Proving Program Properties

Recall the calculation proof method

- Substitution of equals for equals.
- Based on **definitions** or previously proved **theorems**.

- For example consider:

```
(f . g) x = f (g x)
```
(comp)

  - Notice label on equation

- Now prove that composition is associative, i.e.

```
((f . g) . h) x = (f . (g . h)) x
```

  - Can use known equations in either direction.

Example: Proof by calculation

- Pick one side of the equation and transform using rule **comp** above

```
((f . g) . h) x  =
```
by comp (left to right)

```
(f . g) (h x) =
```
by comp (left to right)

```
f (g (h x)) =
```
by comp (right to left)

```
f ((g . h) x) =
```
by comp (right to left)

```
(f . (g . h)) x
```

Example With Regions

- Consider the algebra of Shapes (Ch. 8)
- Suppose we have already proved (Hudak p.100-101):

```
r `Union` Empty = r                              (Axiom 4a)
r `Intersect` univ = r                           (Axiom 4b)
r `Union` Complement r = univ                    (Axiom 5a)
r `Intersect` Complement r = Empty               (Axiom 5b)
r1 `Union` (r2 `Intersect` r3)                   (Axiom 3b)
   = (r1 `Union` r2) `Intersect` (r1 `Union` r3)
```

- Prove: `r `Union` r = r`

```
r =      (by Axiom 4a)
r `Union` Empty = (by 5b)
r `Union` (r `Intersect` Complement r) =   (by 3b)
(r `Union` r) `Intersect` (r `Union` Complement r)=
(r `Union` r)  `Intersect` univ = (by 4b)        (by 5a)^
  r `Union` r
```

Proofs by induction over **finite** lists

- Format over lists

Let P{x} be some proposition (I.e. P{x} :: Bool)

i.e. P is an expression with some free variable x :: [a]

  - x has type :: [a]
  - x may occur more than once in P{x}

e.g.

```
length x = length (reverse x)
all p x => p (head x)
sum (x ++ y) = sum x + sum y
map f (x ++ y) = map f x ++ map f y
(map f . map g) x = map (f . g) x
```

- Then to prove P for all finite lists, we:

  1) Prove P { [] }

  2) Assume P{ xs } and then

     Prove P{ x:xs }

Example: relating map and length

- Definitions and Laws: (These are things we get to assume are true)

```
length [] = 0                          (1)
length (x:xs) = 1 + length xs          (2)
map f [] = []                          (3)
map f (x:xs) = f x: map f xs           (4)
```

- Proposition: (This is what we are trying to prove)

  P{**xs**}: `length (map f xs) = length xs`

- Proof Structure:
  - 1) Prove P{**[]**}:

    `length (map f []) = length []`

  - 2) Assume P{**xs**}: (as well as the definitions and laws)

    `length (map f xs) = length xs`

    Then Prove P{**x:xs**}:

    `length (map f (x:xs)) = length (x:xs)`

Proof

1) Prove: `length (map f []) = length []`

```
length (map f [])  =   (by 3: map f [] = [])
length []
```

2) Assume: `length(map f xs) = length xs`

Prove: `length(map f (x:xs)) = length (x:xs)`

```
length (map f (x:xs)) =
                 (by 4: map f (x:xs) = f x: map f xs)
length (f x:(map f xs)) =
                 (by 2: length (x:xs) = 1 + (length xs) )
1 + length(map f xs) =    (by IH)
1 + length xs  =
                 (by 2: length (x:xs) = 1 + length xs )
length (x:xs)
```

Example: Relating sum and ++

- Definitions and Laws: (These are things we get to assume are true)

```
sum [] = 0                        (1)
sum (x:xs) = x + (sum xs)         (2)
[] ++ ys = ys                     (3)
(x:xs) ++ ys = x:(xs ++ ys)       (4)
```

- Proposition: (This is what we are trying to prove)

  P{`xs`} = `sum (xs ++ ys) = sum xs + sum ys`

  – why do we do induction on the first argument of ++?

- Proof Structure:
  – 1) Prove P{`[]`}:

    sum ([] ++ ys) = sum [] + sum ys

  – 2) Assume P{`xs`}: (as well as the definitions and laws)

    `sum (xs ++ ys) = sum xs + sum ys`

    Then Prove P{`x:xs`}:

    `sum ((x:xs) ++ ys) = sum (x:xs) + sum ys`

Proof

1) Prove: `sum ([] ++ ys) = sum [] + sum ys`

```
sum ([] ++ ys) =          (by 3:  [] ++ ys = ys )
sum ys =                  (arithmetic: 0 + n = n )
0 + sum ys =              (by 1: sum [] = 0 )
sum [] + sum ys
```

2) Assume: `sum (xs ++ ys) = sum xs + sum ys`

Prove: `sum ((x:xs) ++ ys) = sum (x:xs) + sum ys`

sum ((x:xs) ++ ys) = (by 4: (x:xs) ++ ys = x:(xs ++ ys))

sum (x:(xs++ys)) = (by 2: sum (x:xs) = x + (sum xs))

x + sum(xs++ys) = (by IH)

x + (sum xs + sum ys) = (associativity of +: (p + q) + r = p + (q + r))

(x + sum xs) + sum ys = (by 2: sum (x:xs) = x + (sum xs))

sum (x:xs) + sum ys

Proof by induction using Case Analysis

- Prove by induction:

  P{xs} == **`(takeWhile p xs) ++ (dropWhile p xs) = xs`**

- Where:

  (1) [] ++ ys = ys

  (2) (x:xs) ++ ys = x : (xs ++ ys)

  (3) dropWhile p [] = []

  (4) dropWhile p (x:xs) =
      if p x then (dropWhile p xs)
          else x::xs

  (5) takeWhile p [] = []

  (6) takeWhile p (x:xs) =
      if p x then x:(takeWhile p xs)
          else []

Base and Inductive cases

- Base case: P{[]}

  (takeWhile p []) ++ (dropWhile p []) = (by 3,5)

  [] ++ [] = (by 1)

  []

- Induction Step:

  P{ys} => P{y:ys}

Assume:

(takeWhile p ys) ++ (dropWhile p ys) = ys

Prove:

(takeWhile p (y:ys)) ++ (dropWhile p (y:ys)) = (y : ys)

Split Proof

```
(takeWhile p (y:ys)) ++  (dropWhile p (y:ys))  = (by 4,6)

(if p y then y : (takeWhile p ys)
        else []) ++
 (if p y then (dropWhile p ys)
         else y:ys)
```

- Now, either (p y) = True   or   (p y) = False
- So split problem by doing a case analysis

Case 1: Assume: p y = True

(if p y then y : (takeWhile p ys) else []) ++
(if p y then (dropWhile p ys) else y:ys) = (by case assumption)

(y : (takeWhile p ys)) ++ (dropWhile p ys) = (by 2)

y : ((takeWhile p ys) ++ (dropWhile p ys)) = (by I.H.)

y : ys

Case 2: Assume: p y = False

(if p y then y : (takeWhile p ys)else []) ++
(if p y then (dropWhile p ys) else y:ys) = (by case assumption)

[] ++ (y:ys) = (by 1)

y:ys

Structural Induction over Trees

```
data Bintree a = Lf  a
   | (Bintree a) :/\: (Bintree a)
```

» Note all infix constructors start with a colon (:)

- Assume the following definitions and facts:

```
    sumtree :: Bintree a -> Int
(1) sumtree (Lf x) = x
(2) sumtree (a :/\: b) = (sumtree a) + (sumtree b)
    flatten :: Bintree a -> [a]
(3) flatten (Lf x) = [x]
(4) flatten (a :/\: b )=(flatten a) ++ (flatten b)

(5)sum [] =0
(6) sum (x:xs) = x + (sum xs)
(7) Lemma: sum(xs ++ ys) = (sum xs) + (sum ys)
```

Proofs on Trees

To prove a proposition P{t} about all trees t, must prove it for each tree constructor, assuming it is true for all smaller trees.

So, to prove P{t} on a Bintree, we must:

- Prove P{**`Lf x`**}
- Prove that P{**`a`**} && P{**`b`**} => P{**`a :/\: b`**}

Example: Prove P{**`t`**}: **`sum(flatten t) = sumtree t`**

case 1: Prove P{**`Lf x`**}: **`sum(flatten (Lf x)) = sumtree (Lf x)`**

```
sum(flatten (Lf x)) = (by 3: flatten (Lf x) = [x])
   sum [x] =           (by 6: sum (x:xs) = x + sum xs)
   x + (sum []) =      (by 5: sum [] = 0)
   x + 0 =             (by arithmetic: x + 0 = x)
   x =                 (by 1: sumtree(Lf x) = x)
   sumtree (Lf x)
```

Case 2

case 2: Prove P{**a**} && P{**b**} => P{ **a :/\: b**}

Assume: 1) P{**a**}: **sum(flatten a) = sumtree a**

2) P{**b**}:**sum(flatten b) = sumtree b**

Prove: P{**a :/\: b**}: **sum(flatten (a :/\: b))=**
sumtree(a :/\: b)

```
sum(flatten (a :/\: b)) =
```

by 4

```
sum ((flatten a) ++ (flatten b)) =
```

by lemma: 7

```
sum(flatten a) + sum(flatten b)=
```

by I.H. (twice)

```
(sumtree a) + (sumtree b) =
```

by 2

```
sumtree (a :/\: b)
```