Introducing Haskell

from: COS 441 Slides 3B
by David Walker, Princeton

INDUCTIVE PROOFS ABOUT HASKELL PROGRAMS

Recall: Proofs by simple calculation

- Some proofs are very easy and can be done by:
  - unfolding definitions
  - using lemmas or facts we already know
  - folding definitions back up
- Eg:

Definition:
easy x y z = x * (y + z)

given this

we do this proof

Theorem: easy a b c == easy a c b

Proof:

easy a b c

= a * (b + c) (by unfold)

= a * (c + b) (by commutativity of add)

= easy a c b (by fold)

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

case: xs = x:xs'

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys) (LHS of theorem equation)

case: xs = x:xs'

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

```
  length ( [ ] ++ ys )          (LHS of theorem equation)
= length ( ys )                 (unfold ++)
```

case: xs = x:xs'

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)

case: xs = x:xs’

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length -- done, we have RHS)

case: xs = x:xs’

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]       = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)

case: xs = x:xs’

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)

case: xs = x:xs’

length ((x:xs’) ++ ys)	(LHS of theorem equation)
= length (x:(xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys))	(unfold length)

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)

case: xs = x:xs’

length ((x:xs’) ++ ys)	(LHS of theorem equation)
= length (x:(xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys))	(unfold length)

subcase xs’ = [ ]

subcase xs’ = x’:xs’’

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)

case: xs = x:xs’

length ((x:xs’) ++ ys)	(LHS of theorem equation)
= length (x:(xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys))	(unfold length)

subcase xs’ = [ ]

...

subcase xs’ = x’:xs’’

= 1 + length ((x’:xs’’) ++ ys)	(substitution)
= 1 + length (x’:(xs’’ ++ ys))	(unfold ++)
= 1 + 1 + length (xs’’ ++ ys)	(unfold length)

subsubcase xs’’ = [ ] ....

```
[ ] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
length [ ]      = 0
length (x:xs) = 1 + length xs
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof strategy:

- Proof by induction on the length of xs
  - must cover both cases: [ ] and x:xs'
    - apply inductive hypothesis to smaller arguments (smaller lists)
    - In general, Haskell has lots of non-inductive data types like Integers (as opposed to Natural Numbers) so you have to be careful all series of shrinking arguments have base cases
  - use folding/unfolding of Haskell definitions
  - use lemmas/properties you know of basic operations

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:
length ([ ] ++ ys) (LHS of theorem)

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:	
length ([ ] ++ ys)	(LHS of theorem)
= length ys	(unfold ++)
= 0 + (length ys)	(arithmetic)
= (length [ ]) + (length ys)	(fold length)

case done!

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs'

IH: length (xs' ++ ys) = length xs' + length ys

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs'

IH: length (xs' ++ ys) = length xs' + length ys

length ((x:xs') ++ ys) (LHS of theorem)

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs'

IH: length (xs' ++ ys) = length xs' ++ length ys

length ((x:xs') ++ ys)	(LHS of theorem)
= length (x : (xs' ++ ys))	(unfold ++)

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys)	(unfold length)

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys)	(unfold length)
= 1 + (length xs’ + length ys)	(by IH)

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys)	(unfold length)
= 1 + (length xs’ + length ys)	(by IH)
= length (x:xs’) + length ys	(reparenthesizing and folding length)

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys)	(unfold length)
= 1 + (length xs’ + length ys)	(by IH)
= length (x:xs’) + length ys	(reparenthesizing and folding length we have RHS with x:xs’ for xs)

case done!

All cases covered! Proof done!

```
length [ ] = 0
length (x:xs) = 1 + length xs
```

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Exercises

To test your understanding, try to prove the following:

Theorem 1: for all finite lists xs, ys. listSum(xs ++ ys) = listSum xs + listSum ys

```
drop n [ ] = [ ]
drop n (x:xs) = if n <= 0 then x:xs
                else drop (n-1) xs
```

Theorem 2: for all finite lists xs, natural numbers n and m,

drop n (drop m xs) = drop (n+m) xs

Hint: split the inductive case where xs = x:xs into 3 subcases:

case xs = x:xs:

- subcase m = 0 and n = 0: ...
- subcase m = 0 and n = n' + 1 for some natural number n' (ie: n > 0): ...
- subcase m = m'+1 for some natural number m' (ie: m > 0): ...

Summary

- Haskell is
  - a functional language emphasizing immutable data
  - where every expression has a type:
    - Char, Int, (Char, Int, Float), [Int], [[[(Char, [[Int]])]]]
    - Char -> Int, (Char, Char) -> Int -> [(Char, Int)]
    - String = [Char]
- Reasoning about Haskell programs involves
  - substitution of "equals for equals," unlike in Java or C
  - mathematical calculation:
    - unfold function abstractions
    - push symbolic names around like we do in mathematical proofs
    - reason locally using properties of operations (eg: + commutes)
    - use induction hypothesis
    - fold function abstractions back up
- Homework: Install Haskell. Read LYAHFGG Intro, Chapter 1