CS 360: Programming Languages
Lecture 14: Reasoning About Haskell Programs

Geoffrey Mainland

Drexel University

Section 1

Administrivia

Homework 7: Interval Sets

- Homework 7 posted, due Tuesday, March 4.
- 50 points for property-based tests, 50 points for implementation.
- Property-based tests will be evaluated using mutation testing.
- Gradescope autograder (posted later in the week) will check for compilation errors and track time reports.

For Next Week

Please read chapter 10 of *Programming in Haskell*, "Interactive programming."

Final Exam

Final exam will be Thursday, March 20 from 8:00am–10:00am in **Main Auditorium**.

Section 2

Equational Reasoning

Accompanying Reading

Programming in Haskell, 2nd ed.
Chapter 16.

What is equational reasoning?

Equational reasoning is the process of reasoning about programs by substituting one expression for another equal expression.

Expressions are equal when they can be used interchangeably in *any* context without changing the meaning of a program.

What is equational reasoning?

- We are used to using equations to reason about mathematical statements.

$$
\begin{aligned}
xy &= yx && \text{commutativity of multiplication} \\
x + (y + z) &= (x + y) + z && \text{associativity of addition} \\
x(y + z) &= xy + xz && \text{left distributivity of multiplication} \\
(x + y)z &= xz + yz && \text{right distributivity of multiplication}
\end{aligned}
$$

- The same style of reasoning can be used in Haskell, because declarations in Haskell are **definitional**—they specify equalities, not assignments.
- Recall that the substitution model for pure Scheme worked, but only as long as the value of a variable never changed. Values of Haskell variables never change!
- Mathematical equality, $=$, *is not* the same as Haskell's equality operator, (==)!

Equational Reasoning in (Pure) Scheme

```
(define (f x)
  (f x))

(define result (let ((y (f 1)))
                 (if #t
                     1
                     y)))
```

- What is the value of `result`?
- Can we substitute `(f 1)` for `y` in the body of the **`let`** expression?
- What would happen if we went ahead and performed this substitution?
- We must be careful substituting for variables when reasoning about Scheme programs—we could accidentally make a non-terminating program terminate. Even substituting variables with values can change the meaning of a program in the presence of eq?.

Equational Reasoning in Haskell

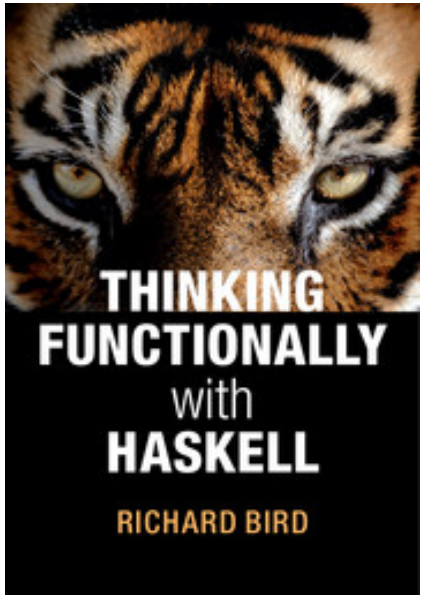
```
f x = f x

result = let y = f 1
         in
           if True then 1 else y
```

- What is the value of `result`?
- Can we substitute `f 1` for `y` in the body of the **`let`** expression?
- In Haskell a **`let`** expression really is an equality! Why? Because of lazy evaluation.
- By the way. . . what is the (most general) *type* of `f`?

Equational Reasoning

- One can reason about imperative code by, e.g., formulating specifications in the predicate calculus and using loop invariants to prove their correctness.
- Haskell is unique in that we can reason about properties of a program directly in the language of its code.
- Efficient algorithms can be *derived* from specifications using equational reasoning.
- 70s: Use algebra to calculate programs.
- Last ≈ 15 years: Use program synthesis to "calculate" programs.

Equational Reasoning



Haskell Equations are Ordered

- We must be careful with Haskell equations, since they are ordered.

```
isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

- Can we replace the expression `isZero n` with **`False`** unconditionally?
- *No*. The equation `isZero n = False` holds *only* when `n /= 0`.

A Simple Proof: Singleton Property of reverse

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Prove that `reverse [x]` $=$ `[x]`

$$
\begin{aligned}
& \texttt{reverse [x]} \\
= \quad & \{\text{list notation}\} \\
& \texttt{reverse (x : [])} \\
= \quad & \{\text{applying reverse}\} \\
& \texttt{reverse [] ++ [x]} \\
= \quad & \{\text{applying reverse}\} \\
& \texttt{[] ++ [x]} \\
= \quad & \{\text{applying ++}\} \\
& \texttt{[x]}
\end{aligned}
$$

An Inductive Proof

```
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

Prove the property $P(\mathsf{n})$ that add n Zero $=$ n.
Question: What should we induct on?
Base case:

$$
\begin{aligned}
& \mathsf{add}\ \mathbf{Zero}\ \mathbf{Zero} \\
= \quad & \{\text{applying add}\} \\
& \mathbf{Zero}
\end{aligned}
$$

So $P(\mathsf{Zero})$ holds.

An Inductive Proof cont'd

```
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

Prove the property $P(\mathsf{n})$ that add n Zero $=$ n.

Inductive case: If $P(\mathsf{n})$ holds, then $P(\mathsf{Succ\ n})$ holds. What is our inductive hypothesis? Our IH is that add n Zero $=$ n.

$$
\begin{array}{lll}
 & & \mathsf{add\ (Succ\ n)\ Zero} \\
 & = & \quad \{\text{applying add}\} \\
 & & \mathsf{Succ\ (add\ n\ Zero)} \\
 & = & \quad \{\text{applying inductive hypothesis}\} \\
 & & \mathsf{Succ\ n}
\end{array}
$$

Exercise: Associativity of Addition

Show that $\forall x, y, z : \texttt{add x (add y z)} = \texttt{add (add x y) z}$.

```
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

Base case:

$$
\begin{aligned}
& \texttt{add Zero (add y z)} \\
= \quad & \{\text{applying outer add}\} \\
& \texttt{add y z} \\
= \quad & \{\text{unapplying add}\} \\
& \texttt{add (add Zero y) z}
\end{aligned}
$$

Exercise: Associativity of Addition cont'd

```
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

Inductive case:

$$
\begin{aligned}
& \texttt{add (Succ x) (add y z)} \\
= \quad & \{\text{applying outer add}\} \\
& \texttt{Succ (add x (add y z))} \\
= \quad & \{\text{inductive hypothesis}\} \\
& \texttt{Succ (add (add x y) z)} \\
= \quad & \{\text{unapplying the outer add}\} \\
& \texttt{add (Succ (add x y)) z} \\
= \quad & \{\text{unapplying the inner add}\} \\
& \texttt{add (add (Succ x) y) z}
\end{aligned}
$$

Structural Induction

- Our proof proceeded by induction on the *structure* of a **Nat**.
- We *did not* induct on the "size" of a **Nat**!
- **Structural induction** allows us to prove properties over an *inductively defined structure*, like a list or a **Nat**.
- Recommended: Show $\forall n \geq 0, \texttt{length (replicate n x)} = \texttt{n}$

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

Exercise: Identity for ++

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Exercise: ++ is associative

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Show `(xs ++ ys) ++ zs` $=$ `xs ++ (ys ++ zs)`. What should we induct on? Hint: `(++)` recurses on its first argument.

Exercise: reverse distributes over ++

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Show `reverse (xs ++ ys) = reverse ys ++ reverse xs`. What should we induct on? Hint: `(++)` recurses on its first argument.
Base case:

```
                reverse ([] ++ ys)
=                 {applying ++}
                reverse ys
=                   {identity for ++}
                reverse ys ++ []
=                 {unapplying reverse}
                reverse ys ++ reverse []
```

Exercise: reverse distributes over ++

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Inductive case:

```
    reverse ((x:xs) ++ ys)
=     {applying ++}
    reverse (x : (xs ++ ys))
=     {applying reverse}
    reverse (xs ++ ys) ++ [x]
=     {induction hypothesis}
    (reverse ys ++ reverse xs) ++ [x]
=       {associativity of ++}
    reverse ys ++ (reverse xs ++ [x])
=     {unapplying the second reverse}
    reverse ys ++ reverse (x:xs)
```

Exercise: `reverse` is its own inverse

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Show `reverse (reverse xs)` $=$ `xs`. We will need the fact we just proved, namely that `reverse` distributes over `++`.

Exercise: Base Case

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
                    reverse (reverse [])
  =                  {applying inner reverse}
                    reverse []
  =                  {applying reverse}
                    []
```

Exercise: Inductive Case

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
        reverse (reverse (x:xs))
=         {applying inner reverse}
        reverse (reverse xs ++ [x])
=           {distributivity of reverse}
        reverse [x] ++ reverse (reverse xs)
=         {singleton list property of reverse}
        [x] ++ reverse (reverse xs)
=         {inductive hypothesis}
        [x] ++ xs
=         {applying ++}
        x:xs
```

Calculating a more efficient `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- How efficient is this version of `reverse`?
- Append is $O(n)$ in the length of the list.
- Each step of `reverse` calls append, so it is $O(n^2)$.
- We want to find a more efficient version of `reverse` that combines the behavior of `reverse` and `++`.
- Idea: Define a *more general* function and *calculate* its implementation (eliminating the dependency on `reverse`).

```
reverse' :: [a] -> [a] -> [a]
reverse' xs ys = reverse xs ++ ys
```

- Idea: Then define `reverse` in terms of the calculated implementation.

```
reverse :: [a] -> [a]
reverse xs = reverse' xs []
```

Calculating a more efficient `reverse`

```
reverse' :: [a] -> [a] -> [a]
reverse' xs ys = reverse xs ++ ys

reverse :: [a] -> [a]
reverse xs = reverse' xs []
```

Goal:

1. Use equational reasoning to *eliminate* (the inefficient) `reverse` from the definition of `reverse'`.
2. Define `reverse` as `reverse' xs []`.

Calculating reverse': Base Case

```
reverse' :: [a] -> [a] -> [a]
reverse' xs ys = reverse xs ++ ys

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
                reverse' [] ys
=                {specification of reverse'}
                reverse [] ++ ys
=                {applying reverse}
                [] ++ ys
=                {applying ++}
                ys
```

Calculating reverse': Inductive Case

```
reverse' :: [a] -> [a] -> [a]
reverse' xs ys = reverse xs ++ ys

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
                        reverse' (x:xs) ys
    =                    {specification of reverse'}
                        reverse (x:xs) ++ ys
    =                    {applying reverse}
                        (reverse xs ++ [x]) ++ ys
    =                     {associativity of ++}
                        reverse xs ++ ([x] ++ ys)
    =                    {specification of reverse'}
                        reverse' xs ([x] ++ ys)
    =                    {applying ++}
                        reverse' xs (x:ys)
```

Calculating reverse'

- This yields the following definitions:

```
reverse' :: [a] -> [a] -> [a]
reverse' []     ys = ys
reverse' (x:xs) ys = reverse' xs (x : ys)

reverse :: [a] -> [a]
reverse xs = reverse' xs []
```

- This implementation is $O(n)$.
- We have calculated the standard accumulating parameter version of reverse.