Lecture 06: Intro to Equational Reasoning; Environments

CS4400 Programming Languages

Readings:

Overview

In the previous lecture we introduced the notion of *bindings*, that is, associating names with a computed value to be re-used later. We also introduced the notion of *substitution* to deal with replacing names with values. Substitution is a syntactic operation: it operates on the whole AST, transforming it in the process. Such transformed AST is the evaluated. Today we will look at another model of associating meaning to names in a program: environments. However, we start with an introduction to the topic of *equational reasoning* on Haskell programs.

Intro to Equational Reasoning with Haskell

Today on "The things you might or might not remember from high-school algebra": algebraic laws. These are equations that specify some properties of arithmetic operations. We can use them to manipulate equations *without changing their meaning*, usually to make the our computations easier.

- commutative laws

$$x + y = y + x$$
$$x \cdot y = y \cdot x$$

- associative laws

$$x + (y + z) = (x + y) + z$$
$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

- distributive law

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

Side note. these laws can be proven from these axioms for addition and multiplication:

$$x + 0 = x$$
$$x + (y + 1) = (x + y) + 1$$

$$x \cdot 0 = 0$$
$$x \cdot (y + 1) = x + (x \cdot y)$$

Using these laws (and the definition of ^{2} and multiplication), we can, for example, show that $(x + y)^2 = x^2 + 2xy + y^2$:

$$
\begin{array}{ll}
 & (x + y)^2 \\
= & \qquad \{ \text{ definition of squaring } \} \\
 & (x + y) \cdot (x + y) \\
= & \qquad \{ \text{ distributivity } \} \\
 & (x + y) \cdot x + (x + y) \cdot y \\
= & \qquad \{ \text{ commutativity (twice) } \} \\
 & x \cdot (x + y) + y \cdot (x + y) \\
= & \qquad \{ \text{ distributivity (twice) } \} \\
 & (x \cdot x + x \cdot y) + (y \cdot x + y \cdot y) \\
= & \qquad \{ \text{ associativity } \} \\
 & x \cdot x + (x \cdot y + (y \cdot x + y \cdot y)) \\
= & \qquad \{ \text{ associativity } \} \\
 & x \cdot x + ((x \cdot y + y \cdot x) + y \cdot y) \\
= & \qquad \{ \text{ commutativity } \} \\
 & x \cdot x + ((x \cdot y + x \cdot y) + y \cdot y) \\
= & \qquad \{ \text{ definition of } ^2 \text{ (twice) } \} \\
 & x^2 + ((x \cdot y + x \cdot y) + y^2) \\
= & \qquad \{ \text{ definition of multiplication } \} \\
 & x^2 + ((2 \cdot (x \cdot y)) + y^2) \\
= & \qquad \{ \text{ conventions for () } \} \\
 & x^2 + 2 \cdot x \cdot y + y^2
\end{array}
$$

Why is this interesting? Well, some of these laws have implications for computational efficiency. For example, consider the difference between

```
x * y + x * z
```

and

```
x * (y + z)
```

They compute the same result, but which one is more efficient? Which one would you prefer to use in a loop that runs 10^{14} times?

The big idea here is, that we can use the same style of reasoning for Haskell programs. And Haskell is particularly suitable for this style of reasoning about programs.

Any definition in Haskell can be read as an equation. E.g.,

```
double :: Integer -> Integer
double x = x + x
```

is a definition, but the defining equation also gives us a *property* (in this case an *axiom*). Whenever we see `double x` (where `x` is an arbitrary expression), we can replace it with the right hand side `x * x`. Not only that: we can also go in the opposite direction and replace `x * x` with `double x`.

We have to be careful, however, because the order of equations is significant in Haskell, so we might not be always able to simply replace whatever matched. Consider:

```
isZero :: Integer -> Boolean`
isZero 0 = True
isZero n = False
```

Replacing `isZero 0` with `True` and vice-versa is always OK. However, `isZero n` is not valid for arbitrary `n`, only if `n` $\neq$ `0`. We have to be careful with patterns that are *overlapping*. The above can be rewritten to an explicit equivalent with guards:

```
isZero :: Integer -> Boolean`
isZero 0 = True
isZero n | n /= 0 = False
```

Now we made the patterns *non-overlapping*. In math this would be something like this:

$$\begin{aligned}\mathsf{isZero}(0) &= \mathit{true}\\ \mathsf{isZero}(n) &= \mathit{false} \quad \mathit{if}\ n \neq 0\end{aligned}$$

Example: Show that the reverse of a singleton list is the list itself.

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

Trying to show: `reverse [x] = x`.

```
  reverse [x]
=          { syntactic sugar for [x] }
  reverse (x : [])
=          { definition of reverse }
  reverse [] ++ [x]
=          { definition of reverse }
  [] ++ [x]
=          { definition of (++)[1] }
  [x]
```

Another example: `not (not x)` = x for any (boolean) x.

Definition:

```
not :: Bool -> Bool
not True = False
not False = True
```

We don't know what x is, so we need to consider all cases (*case analysis*).

Consider x = `True`:

```
  not (not True)
=          { definition of not }
  not False
=          { definition of not }
  True
```

[1] The operator (++) (list append/concatenation) is defined in the Prelude as follows:

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

Now consider x = False:

```
  not (not False)
=         { definition of not }
  not True
=         { definition of not }
  False
```

When we revisit equational reasoning again, we will look at performing induction to show more interesting properties for numbers or lists.

But we're doing programming languages and we discussed how us defining an interpreter in Haskell means that we are relating the meaning our object PL to the meaning of Haskell. So, as another example, let's use our simple SAE interpreter to show that our Add operation inherits associativity from Haskell:

```
eval :: SAE -> Integer
eval (Number n)
eval (Add e1 e2) = eval e1 + eval e2
...
```

We want to show that `eval (Add e1 (Add e2 e3)) = eval (Add (Add e1 e2) e3)`.

```
  eval (Add e1 (Add e2 e3))
=         { definition of eval }
  eval e1 + eval (Add e2 e3)
=         { definition of eval }
  eval e1 + (eval e2 + eval e3)
=         { associativity of +[2] }
  (eval e1 + eval e2) + eval e3
=         { definition of eval }
  eval (Add e1 e2) + eval e3
=         { definition of eval }
  eval (Add (Add e1 e2) e3)
```

Note, that a Maybe output type complicates this picture somewhat, but we can still make it work.

Where can this go wrong in PL?

In math, we expect $2x = x + x$ to apply universally. How about programming languages? Scheme (or BSL)? Consider:

[2] See "Numeric type classes" at https://hackage.haskell.org/package/base-4.14.0.0/docs/Prelude.html. Okay, what's the big deal with equational reasoning in Haskell?

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

Question: Does `(* 2 (f 20)) = (+ (f 20) (f 20))` hold?

How about:

```
(define (g x)
  (begin
    (write x)
    (+ x x)))
```

or similarly in Java:

```
private int g(int x) {
  System.out.println(x);
  return x + x;
}
```

or, again in Java:

```
int c = 0;

private int g(int x) {
  c++;
  return x + x;
}
```

Question: Does `(* 2 (g 20)) = (+ (g 20) (g 20))` (or 2 `* g(x) = g(x) + g(x)`) hold?

In many programming languages, the possibility of *side-effects* (printing, mutating a variable, setting off a nuclear bomb, ...) breaks this. Many programming languages do not have *referential transparency*:

> An expression is called *referentially transparent* if it can be replaced with its corresponding value without changing the program's behavior.[3]

On the other hand, how would we write the above example in Haskell? Let's start with `f`:

[3] John C. Mitchell (2002). Concepts in Programming Languages. Cambridge University Press

```
f x = x + x
```

Here, `2 * f x` = `f x + f x` definitely holds. Just like in Scheme. No issue here.

How do we define an equivalend of `g`?

```
g x = ... putStrLn (show x) ...x + x
```

What do we replace `...` with? First of all, what is the type of `g`? For `f`, it is `Integer -> Integer` (more generally `Num a => a -> a`). For `g`, however, we need to go into `IO`!

```
g x = do
    putStrLn (show x)
    return (x + x)
```

For now, think of `return` exclusively as a way of returning values from `IO` computations and don't use them anywhere else but `IO` functions that also return a value. We'll find more uses of them later. Well this means that the type of `g` is

```
g :: Integer -> IO Integer
```

Now, a function in with this type cannot be even simply used in an expression: `2 * g x` results in a type error. We can only use `g` inside of another `IO` computation!

```
do
    y <- g x
    ... (2 * y) ...
```

We are forced to make things very explicit:

1. *Run* the computation `g x`
2. Retrieve the value
3. Use the value in a computation.

There is no other way.

The separation of *pure* and "impure" code gives us guarantees. When we do see `2 * f x`, we can really replace it with `f x + f x`.

Bindings with Environments

Now that we equipped ourselves with the basics of equational reasoning, let's return to bindings. We used substitution: walk the AST and replace every occurrence of the given variable with the given value. But we are potentially doing extra work: what if some variable references are never reached? E.g.:

```
(let (x (+ 11 22)) (* 12 32))
```

Or a more obscured one:

```
(let (x (* 24 15))
     (+ (/ 20 (- x x))
        (+ x (/ x 2))))
```

Here, not all x are needed. Why?

We can use something like a cache of substitutions. Instead of eagerly substituting *all* variables as soon as we encounter a let, we remember what x was and continute evaluations until we actually need the value of x.

How do we remember what the value associated with a variable was? A map, i.e., a set of mappings between names (keys) and values. How do we represent a map? There are multiple choices: hashmap, association lists, binary search trees, . . .

But. . . these are all the same to us. We can choose whichever we want. All we need is that they satisfy a few axioms. Let's say that a map that has no mappings is called empty. Let's say that we have an operation which adds a mapping to a given map and returns an updated version, called set. And let's say that we have an operation, get, which, given a key and a map, finds the corresponding value. These operations should satisfy the following:

```
get x (set x v m) = v
get x (set y v m) = get x m, if x /= y
```

Additionally, we might say that get x empty is undefined for any x.

Let's defer defining an actual representation until later and just go ahead and program an environment-based evaluator. However, let us sketch the types of get, set, empty:

```haskell
type Map k v
type Env = Map Variable Integer

get :: Variable -> Env -> ???

set :: Variable -> Value -> Env -> Env

empty :: env
```

Continued in `Lec06.hs` available on the website. These notes might be updated and extended at a later date, based on questions and comments.