# Abstraction Functions

Sebastiaan Joosten

proof properties in
exams

University of Minnesota

# Motivation

- We want to consider things equal that are not:
  - ([1;2;3],[]) '=' ([],[3;2;1]) in the case of queues
  - {Joosten: 3, Moen: 10} = {Moen: 10, Joosten: 3, Joosten: 3} for dictionaries
  - 4/8 = 1/2 = -1/-2 if we're representing fractions
  - … etc
- We wish to define a function 'eq' that satisfies:
  - eq a b = eq b a
  - if eq a b && eq b c then eq a c else true
  - eq a a
  - If 'eq a b' then 'a' and 'b' are *behaviorally* the same

# Canonical forms

- Most data types can be put into a canonical form:
  - ([1;2;3],[]) ‘=’ ([],[3;2;1]) in the case of queues
    - Let (a,[]) be the canonical form.
  - 4/8 = 1/2 = -1/-2 if we’re representing fractions
    - Let a/b be canonical if the gcd of a and b is 1 and b is positive.
- If ‘cf’ is a function that defines a canonical form, then we can define ‘eq’ as follows:
  - let eq x y = (cf x = cf y)

# Properties of ‘eq’

- eq a b = eq b a
- if eq a b && eq b c then eq a c else true
- eq a a
- If ‘eq a b’ then ‘a’ and ‘b’ are *behaviorally* the same

- Let eq x y = (cf x = cf y)

- This already takes care of the first three properties.
- Let’s see the proofs!!

# Properties of ‘eq’

- Let eq x y = (cf x = cf y)
- We prove: eq a b = eq b a
- eq a b
  = {def eq}
  (cf a = cf b)
  = {(a = b) = (b = a)}
  (cf b = cf a)
  = {def eq}
  eq b a

# Properties of ‘eq’

- Let eq x y = (cf x = cf y)
- We prove: if eq a b, and eq b c, then eq a c
- Lemma: (cf a = cf b)
  - Proof:
    (cf a = cf b)
    = {def eq}
    eq a b
    = {explicit assumption}
    true
- Similarly: (cf b = cf c)
  - (proof as above)
- Lemma 3: (cf a = cf c)
  - cf a
    = {lemma 1}
    cf b
    = {lemma 2}
    cf c
- Final proof:
  - eq a c
    = {def eq}
    (cf a = cf c)
    = {Lemma 3}
    true

# Properties of ‘eq’

- Let eq x y = (cf x = cf y)
- We prove: if eq a b, and eq b c, then eq a c
- Lemma: (cf a = cf b)
  - Proof:
    (cf a = cf b)
    = {def eq}
    eq a b
    = {explicit assumption}
    true
- Similarly: (cf b = cf c)
  - (proof as above)

- Lemma 3: (cf a = cf c)
  - cf a
    = {lemma 1}
    cf b
    = {lemma 2}
    cf c
- Final proof:
  - eq a c
    = {def eq}
    (cf a = cf c)
    = {Lemma 3}
    true

Note about this proof: we’re using that ‘(L = R) = true’ iff ‘L = R’ in three places!

# Properties of 'eq'

- eq a a
  = {def eq}
  (cf a = cf a}
  = {(a=a) = true}
  true
- This completes the proofs of these properties:
  - eq a b = eq b a
  - if eq a b && eq b c
    then eq a c else true
  - eq a a

# Final property:

- If ‘eq a b’ then ‘a’ and ‘b’ are *behaviorally* the same

- This depends on:
  - how cf is defined
  - how ‘behaviorally’ is defined

- We’ll say that ‘a’ and ‘b’ are behaviorally the same if we cannot run any test that could tell them apart.
- The idea is called ‘congruence’. Let’s see what this idea looks like in practice first!

# In practice

- Let’s take a look at a module for rationals

- With this signature:
  - type t
  - is_zero : t -> bool
  - plus : t -> t -> t
  - make : int -> int -> t

- And this implementation
  - type t = int * int
  - make a b = (a,b)
  - cf (a,b)
    = let g = gcd a b in
    (a / g, b / g)
  - plus (a,b) (c,d)
    = (a*d+b*c, b*d)
  - is_zero (a,b) = (a = 0)

# ‘Behaves the same’

- The only way to show that two numbers are different (under this signature) is to use is_zero

- let x = make 1 2
- let y = make 2 4
- let z = make 1 4

- is_zero (plus (make (-1) 4) x) = false
- is_zero (plus (make (-1) 4) y) = false
- is_zero (plus (make (-1) 4) z) = true

# ‘Behaves the same’

- The only way to show that two numbers are different (under this signature) is to use is_zero

- let x = make 1 2
- let y = make 2 4
- let z = make 1 4

- is_zero (plus (make (-1) 2) x) = true
- is_zero (plus (make (-1) 2) y) = true
- is_zero (plus (make (-1) 2) z) = false

# Showing that 'cf' is okay…

- cf (a,b)
  = let g = gcd a b in
  (a / g, b / g)

- Let's take a close look at is_zero…
- Suppose that cf (a,b) = cf (c,d)
- Does that mean that they behave the same wrt is_zero?
- We first (somewhat informally) prove that: is_zero (cf x) = is_zero x:
- is_zero (cf (a,b))
  = {definition of cf with the let applied all in one step}
  is_zero (a / g, b / g)
  = {definition of is_zero}
  (a / g = 0)
  = {some reasoning: g is a divisor of a, so a / g = 0 iff a = 0}
  (a = 0)
  = {definition of is_zero}
  is_zero (a, b)

# Proof that ‘cf’ is okay for is_zero

- Let’s assume we have two equivalent rationals:
- cf a = cf b
- We’ll use the property from the previous slide:
  is_zero (cf x) = is_zero x, we get:
-  is_zero a
  = {property}
   is_zero (cf a)
  = {cf a = cf b, the rationals are equivalent}
   is_zero (cf b)
  = {property}
   is_zero b

# What about ‘plus’?

- It’s nice that ‘is_zero’ cannot tell two equal rationals apart, but perhaps we can turn two equal rationals into different rationals by adding something clever:
- cf a = cf b  (a and b have the same normal form)
- For some clever choice of x we get:
  is_zero (plus a x) ≠ is_zero (plus b x)
- for that to work, we’d need that:
- cf (plus a x) ≠ cf (plus b x)
- Let’s prove that this does not happen!

# ‘cf’ is okay for plus!

- Assume:
  cf a1 = cf a2
  cf b1 = cf b2
- Show:
  cf (plus a1 b1) = cf (plus a2 b2)
- We’ll not do this proof here
  (requires more reasoning about divisors)

# What about ‘numerator’

- let numerator (a,b) = a

- Is cf okay?

# What about ‘numerator’

- let numerator (a,b) = a

- Is cf okay?

- cf (1,2) = (1,2)
- cf (2,4) = (1,2)
- numerator (1,2) = 1
- numerator (2,4) = 2

- No! This is not okay…

# What about ‘numerator’

- let numerator x = let (a,b) = cf x in a

- Is cf okay now?

- cf (1,2) = (1,2)
- cf (2,4) = (1,2)
- numerator (1,2) = 1
- numerator (2,4) = …

# What about ‘numerator’

- let numerator x = let (a,b) = cf x in a

- Is cf okay now?

- cf (1,2) = (1,2)
- cf (2,4) = (1,2)
- numerator (1,2) = 1
- numerator (2,4) = 2
- yay!

# Congruence: intuition

- Suppose we define a module with abstract type t
- Let f : t -> x be a function in the module, where x is some type that we can inspect outside the module.
- If ‘eq a b’, then we require: f a = f b

- Now let f : t -> t
- If ‘eq a b’, then we just require ‘eq (f a) (f b)’

- We saw a relation that is like eq on t and like = on other data types last lecture. This is precisely the relation we need!

# Congruence: defined

- We define ≡ as follows:
  - x ≡ y = eq x y for 'eq' as defined in our module
  - x ≡ y = x = y for x, y : int, float, … (basic built-in type)
    (a, b) ≡ (c, d) = (a ≡ c && b ≡ d)
    Some a ≡ None = false
    Some a ≡ Some b = (a ≡ b)
    (.. and so on ..)
- We say that ≡ is a congruence for a function f if:
  x ≡ y means that f x ≡ f y.
- We say that ≡ is a congruence for a module if it is a congruence for every function exposed by the module.

# What does this mean for cf?

- This requirement can be changed a bit:

  if $x \equiv y$ then: $f\ x \equiv f\ y$

- The 'canonical form' is often a form of type t as well, that 'behaves the same'. This means for f we have:

  $f\ (cf\ x) \equiv f\ x$

- We can use this to prove:
- $f\ x$

  $\equiv$ {property}

  $f\ (cf\ x)$

  $=$ {$x \equiv y$, so: $cf\ x = cf\ y$)

  $f\ (cf\ y)$

  $\equiv$ {property}

  $f\ y$

# What about sets

- Sets are often represented by trees, not lists
- However, a good canonical representation of a set, would be a sorted list with no duplicate elements
- (It would be unnecessary to turn that back into a tree)
- This means that our canonical representation of sets of type ‘a would be of type: ‘a t -> ‘a list

- We’ll call such a function “abstraction function”, or ‘af’. (to match the textbook).

- Aside: The mathematical term for these function ‘af’ and ‘cf’ (provided that ‘eq’ is a congruence) is called ‘homomorphism’

# What’s the difference between cf and af?

- Other than the type, the canonical-form function typically satisfies:
- cf (cf x) = cf x
- (taking the canonical form of what is already canonical, does not change it)
- That allows us to prove something like:
  is_zero (cf x) = is_zero x
  which is a sufficient condition for eq to be a congruence on is_zero.

- Why does af not satisfy this property?

# Do we have to write an abstraction function?

- Reasons to write the abstraction function:
  - Often quite easy to write
  - It’ll be clear what is equivalent and what is not
  - Proving and testing that it is a congruence can expose hard-to-find bugs
- Reasons not to write it:
  - Nobody might run it outside of tests (bad reason imho: nobody runs comments either)
  - It’s not part of the assignment

# Another useful function

- Here's another trick ocaml programmers use is to write a representation invariant:

- module Ratio = struct
-   type t = int * int
-   let invariant (a, b) =
-     assert b > 0;
-     assert (gcd a b = 1)
-   …
- end

- We'll cover it next week!

# Next lecture

- We'll go over a sample midterm again
- Keep an eye on Canvas