15-150 Equational Reasoning Guide

February 21, 2012

In ASCII, we'll write `==` for equivalence. `e1 == e2 : T` is a predicate on two closed expressions that have the same type `T`. In math, we'll write $e_1 \cong e_2 : \tau$. But we'll usually suppress the type, and unless otherwise stated, we assume that the two terms are well-typed.

The intuition for equivalence is that two expressions are equivalent iff they both evaluate to the same value, or they both go into an infinite loop, or they both raise the same exception. The rules we give here are sound in the presence of non-termination and exceptions, but they don't let us reason about exceptions in detail; look for an update later in the semester.

1 Definitions

- `e value` means that `e` is a value; for each type we say what the values are. This includes numbers, strings, applications of datatype constructors to values (e.g. `v :: vs` where `v` and `vs` are values), functions (`fn x => whatever`), etc.
- `e valuable` means "there exists a `v` such that `v value` and `e == v`".
- For `f : T2 -> T`, `f total` means "for all values `v2 : T2`, if `v2 value`, then `(f v2) valuable`".
- We write "for all values `e : T` ..." or even "for all `v : T` ..." to mean "for all `e : T`, if `e value` then ...". This means that a quantifer over arbitary expressions must be explicitly indicated as "for all expressions `e : T` ...".
- We write `[e/x]e'` for the substutition of `e` in for `x` in `e'`.

2 Equivalence Relation

Equivalence is an equivalence relation:

```
e == e

e2 == e1 if e1 == e2

e1 == e3 if e1 == e2 and e2 == e3
```

3 Congruence

Equivalence is a congruence, which means we can replace one expression with an equivalent one "inside" any bigger expression. For example:

```
e1 e2 == e1 e2'  if  e2 == e2'
e1 e2 == e1' e2  if  e1 == e1'

(e1, e2) == (e1, e2')  if  e2 == e2'
(e1, e2) == (e1', e2)  if  e1 == e1'

c e == c e'  if  e == e' [c is a datatype constructor]

let val x = e1 in e2 == let val x = e1' in e2   if  e1 == e1'

case e of [] => e1  | x :: xs => e2 ==
case e of [] => e1' | x :: xs => e2     iff e1 == e1'
```

For positions that go under binders, we quantify over all values of the appropriate type:

```
(fn x : T => e) == (fn x : T => e')  if
  for all values v : T, [v/x]e == [v/x]e'

(let val x : T = e1 in e2) == (let val x : T = e1 in e2')   if
  for all values v : T, [v/x]e2 == [v/x]e2'

case e of [] => e1 | x :: xs => e2 ==
case e of [] => e1 | x :: xs => e2'    if
  for all values v and vs, [v/x][vs/xs]e2 == [v/x][vs/xs]e2'
```

And similarly for other kinds of `case`.

4 Valuable Reductions

ML's operational semantics are call-by-value: you evaluate the argument to a function before substituting into the body. This is expressed by the following rules:

```
(fn x => e) e2 |-> (fn x => e) e2'  if  e2 |-> e2'
(fn x => e) e' |-> [e'/x] e  if  e' value
```

Because of this evaluation order, it's unsound to take the second step rule as an equation without the value condition. For example, `(fn _ => 6) loop` (which loops) is not equal to `6`, but the rule

```
(fn x => e) e' == [e'/x]e
```

would equate these, because `[loop/x]6 == 6`.

It is certainly OK to take each step rule as an equivalence, with the same `value` premises as the step rule itself:

```
(fn x => e) e' == [e'/x]e  if  e' value
(case [] of [] => e1 | x :: xs => e2) == e1
(case v::vs of [] => e1 | x :: xs => e2) == [v/x][vs/xs]e2 if v::vs value
(let val (x,y) = (v1,v2) in e end) == [v1/x][v2/y]e if (v1,v2) value
```

However, with these rules, you always need to reduce the argument to a function to a value before we can reduce. For example, to show that `map (g o f) == map g o map f` (for total `f` and `g`), we get stuck at `map g (f x :: map f xs)`—we'd like to reduce this to the expression `g (f x) :: map g (map f xs)` but can't because `(f x :: map f xs)` is not a value.

However, we can reason as follows: because equivalence is a congruence, if an expression *has a value*, then a function applied to it reduces. We can prove this by replacing the expression with its value, doing the reduction, and then replacing it back. This insight leads to the following rules:

```
(fn x => e) e' == [e'/x]e  if  e' valuable
(case v::vs of [] => e1 | x :: xs => e2) == [v/x][vs/xs]e2  if  v::vs valuable
(let val (x,y) = (v1,v2) in e end) == [v1/x][v2/y]e   if  (v1,v2) valuable
```

Here `e' valuable` means there is some value that it's equivalent to (see above). Thus, the fact that `e'` has a value is used to satisfy a *premise*, but we don't need to use that value in the rest of the equational deduction.

As a special case, we have that

```
if   e |-> v
then e == v
```

5 Type-Directed Rules

Two functions are equal if they agree on all arguments:

```
f == g : T2 -> T if
  for all values v : T2, f v == g v : T
```

Note: because equivalence is a congruence, the premise is interprovable with

```
for all values v and v', if v == v' : T2, then f v == g v' : T
```

Constructors are injective and disjoint:

```
true != false
```

```
x::xs != []
if x::xs == y::ys then x == y and xs == ys
```

(analogously for other datatypes)

6 Valuability

As lemmas, we can show that valuability has the same rules as values for the intro forms:

```
[] valuable
v :: vs valuable if x and xs are valuable
(v1,v2) valuable if v1 valuable and v2 valuable
(fn x => e) valuable
```

The advantage of valuability, relative to values, is that elimination forms are valuable under appropriate conditions:

```
f e valuable if f total and e valuable

(case e of [] => e1 | x::xs => e2) valuable if
   e valuable
   e == [] implies e1 valuable
   for all values v,vs, e == v::vs implies [v/x][vs/xs]e2 valuable
```

and similarly for `let` and other kinds of `case`s.

In general, valuability is closed under equivalence:

```
e valuable if e == e' and e' valuable
```

7 Inversions

For closed terms, `e == v` implies `e ==> v'` where `v' == v`. This justifies the following inversions, which are sometimes useful for backwards reasoning:

```
if   e1 e2 == v
then there exists an expression e and a value v such that
     e1 == (fn x => e)
     e2 == v2
     [v2/x]e == v

if   case e of ... is valuable
then e             is valuable
```