

CSE341: Programming Languages Lecture 12 Equivalence

Dan Grossman Spring 2019

Last Topic of Unit

More careful look at what "two pieces of code are equivalent" means

- Fundamental software-engineering idea
- Made easier with
 - · Abstraction (hiding things)
 - · Fewer side effects

Not about any "new ways to code something up"

Spring 2019

CSE341: Programming Languages

2

Equivalence

Must reason about "are these equivalent" all the time

- The more precisely you think about it the better
- · Code maintenance: Can I simplify this code?
- Backward compatibility: Can I add new features without changing how any old features work?
- · Optimization: Can I make this code faster?
- · Abstraction: Can an external client tell I made this change?

To focus discussion: When can we say two functions are equivalent, even without looking at all calls to them?

- May not know all the calls (e.g., we are editing a library)

Spring 2019

CSE341: Programming Languages

A definition

Two functions are equivalent if they have the same "observable behavior" no matter how they are used anywhere in any program

Given equivalent arguments, they:

- Produce equivalent results
- Have the same (non-)termination behavior
- Mutate (non-local) memory in the same way
- Do the same input/output
- Raise the same exceptions

Notice it is much easier to be equivalent if:

- There are fewer possible arguments, e.g., with a type system and abstraction
- We avoid side-effects: mutation, input/output, and exceptions

Spring 2019

CSE341: Programming Languages

Example

Since looking up variables in ML has no side effects, these two functions are equivalent:

fun f
$$x = x + x$$



val y = 2
fun f x = y * x

But these next two are not equivalent in general: it depends on what is passed for ${\bf f}$

– Are equivalent if argument for ${\tt f}$ has no side-effects

$$fun g (f,x) = (f x) + (f x)$$



- Example: g ((fn i => print "hi" ; i), 7)
- Great reason for "pure" functional programming

Spring 2019

CSE341: Programming Languages

Another example

These are equivalent *only if* functions bound to g and h do not raise exceptions or have side effects (printing, updating state, etc.)

- Again: pure functions make more things equivalent



fun f x =
 let
 val z = h x
 val y = g x
 in
 (y,z)
 end

- Example: g divides by 0 and h mutates a top-level reference
- Example: g writes to a reference that h reads from

Spring 2019

CSE341: Programming Languages

One that really matters

Once again, turning the left into the right is great but only if the functions are pure:

map f (map g xs)

map (f o g) xs

Spring 2019

CSE341: Programming Languages

Syntactic sugar

Using or not using syntactic sugar is always equivalent

- By definition, else not syntactic sugar

Example:

But be careful about evaluation order

Spring 2019

CSE341: Programming Languages

Standard equivalences

Three general equivalences that always work for functions

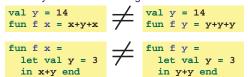
- In any (?) decent language

1. Consistently rename bound variables and uses

val
$$y = 14$$

fun $f x = x+y+x$ val $y = 14$
fun $f z = z+y+z$

But notice you can't use a variable name already used in the function body to refer to something else



Spring 2019 CSE341: Programming Languages

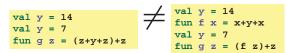
Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

2. Use a helper function or do not

But notice you need to be careful about environments



Spring 2019

CSE341: Programming Languages

10

12

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language
- 3. Unnecessary function wrapping

But notice that if you compute the function to call and *that computation* has side-effects, you have to be careful

Spring 2019

CSE341: Programming Languages

11

One more

If we ignore types, then ML let-bindings can be syntactic sugar for calling an anonymous function:

let val
$$x = e1$$
 (fn $x => e2$) e1 in e2 end

- These both evaluate e1 to v1, then evaluate e2 in an environment extended to map x to v1
- So exactly the same evaluation of expressions and result

But in ML, there is a type-system difference:

- \mathbf{x} on the left can have a polymorphic type, but not on the right
- Can always go from right to left
- If x need not be polymorphic, can go from left to right

pring 2019 CSE341: Programming Languages

What about performance?

According to our definition of equivalence, these two functions are equivalent, but we learned one is awful

(Actually we studied this before pattern-matching)

```
fun max xs =
                            fun max xs =
  case xs of
                               case xs of
    [] => raise Empty
                                [] => raise Empty
    x::[] => x
                               x::[] => x
   x::xs' =>
                               x::xs' =>
       if x > max xs'
                                  let
       then \mathbf{x}
                                    val y = max xs'
       else max xs'
                                  in
                                    if x > y
                                     then x
                                    else y
                                  end
                  CSE341: Programming Languages
Spring 2019
```

Different definitions for different jobs

- PL Equivalence (341): given same inputs, same outputs and effects
 - Good: Lets us replace bad max with good max
 - Bad: Ignores performance in the extreme
- · Asymptotic equivalence (332): Ignore constant factors
 - Good: Focus on the algorithm and efficiency for large inputs
 - Bad: Ignores "four times faster"
- Systems equivalence (333): Account for constant overheads, performance tune
 - Good: Faster means different and better
 - Bad: Beware overtuning on "wrong" (e.g., small) inputs; definition does not let you "swap in a different algorithm"

Claim: Computer scientists implicitly (?) use all three every (?) day

Spring 2019

CSE341: Programming Languages

14