

CSE341: Programming Languages Lecture 12 Equivalence

> **Brett Wortzman** Spring 2020

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 2020

CSE 341: Programming Languages

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)

CSE 341: 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

CSE 341: Programming Languages

Example

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

fun
$$f x = x + x$$

val v = 2fun f x = y * x

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

- Are equivalent if argument for f has no side-effects

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

 $(f x) + (f x)$



val y = 2fun g (f,x) =y * (f x)

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

CSE 341: 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.)

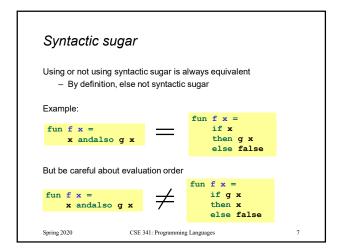
- Again: pure functions make more things equivalent

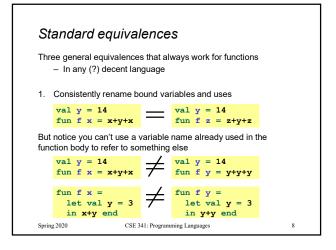


fun f x = val z = h xval y = g xin (y,z) end

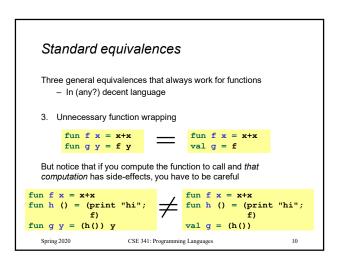
- Example: q divides by 0 and h mutates a top-level reference
- Example: g writes to a reference that h reads from

CSE 341: Programming Languages





```
Standard equivalences
Three general equivalences that always work for functions
   - In (any?) decent language
2. Use a helper function or do not
                               val y = 14
fun g z = (z+y+z)+z
                               fun f x = x+y+x
                               fun g z = (f z) + z
But notice you need to be careful about environments
                                val v = 14
                                fun f x = x+y+x
 val y = 7
                                val y = 7
 fun g z = (z+y+z)+z
                                fun g z = (f z)+z
                   CSE 341: 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 x
                                      val y = max xs'
       else max xs'
                                    in
                                      if x > y
                                      then x
                                      else v
                   CSE 341: Programming Languages
                                                        12
Spring 2020
```

Different definitions for different jobs

- PL (Functional) 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 2020

CSE 341: Programming Languages

13