## **VAUL G. ALLEN SCHOOL of computer science & engineering**

# 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"

# 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)

# 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

#### Example

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

fun f x = x + xwal y = 2fun 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 **f** has no side-effects

- Example: g ((fn i => print "hi" ; i), 7)

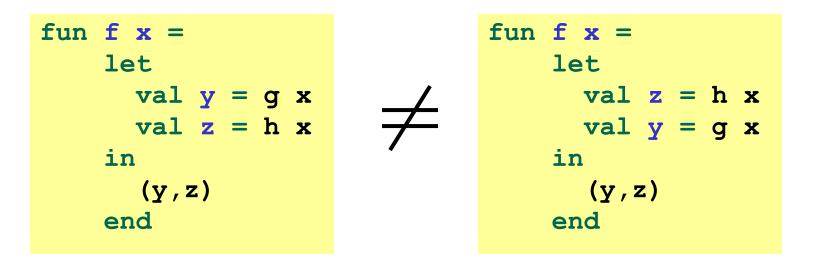
- Great reason for "pure" functional programming

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

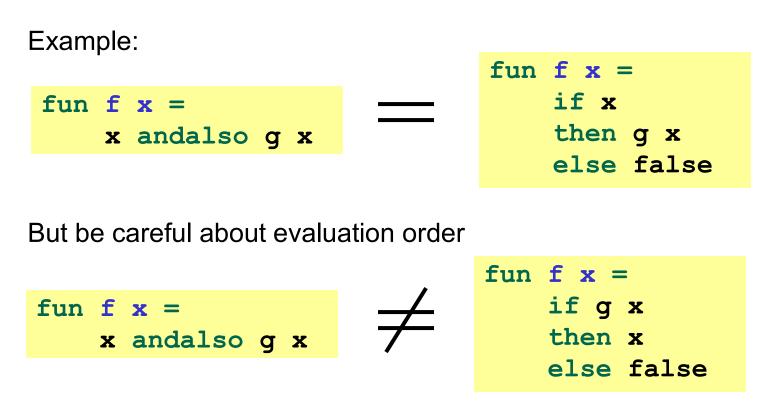


- Example: g divides by 0 and h mutates a top-level reference
- Example: g writes to a reference that h reads from

## Syntactic sugar

Using or not using syntactic sugar is always equivalent

– By definition, else not syntactic sugar



## Standard equivalences

Three general equivalences that always work for functions

- In any (?) decent language
- 1. Consistently rename bound variables and uses

val y = 14val y = 14fun f x = x+y+xfun 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

$$\begin{array}{c} \text{val } y = 14 \\ \text{fun f } x = x + y + x \end{array} \qquad \checkmark \qquad \begin{array}{c} \text{val } y = 14 \\ \text{fun f } y = y + y + y \end{array}$$

$$\begin{array}{c} \text{fun f } x = \\ \text{let val } y = 3 \\ \text{in } x + y \text{ end} \end{array} \qquad \checkmark \qquad \begin{array}{c} \text{fun f } y = \\ \text{let val } y = 3 \\ \text{in } y + y \text{ end} \end{array}$$

CSE 341: Programming Languages

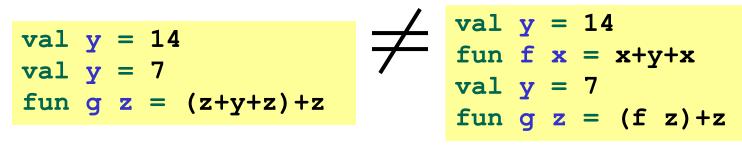
## Standard equivalences

Three general equivalences that always work for functions

– In (any?) decent language

2. Use a helper function or do not

But notice you need to be careful about environments



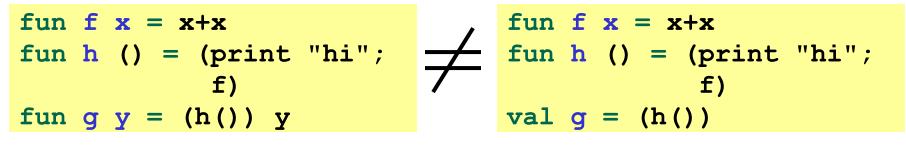
## Standard equivalences

Three general equivalences that always work for functions

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

fun f x = x+xfun f x = x+xfun g y = f yval g = f

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



#### 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

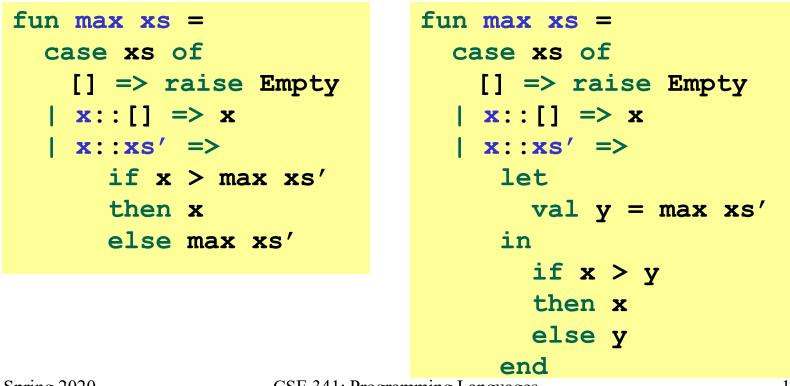
But in ML, there is a type-system difference:

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

#### 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)



# 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