PAUL G. ALLEN SCHOOL of computer science & engineering	Last Topic of Unit More careful look at what "two pieces of code are equivalent" means – Fundamental software-engineering idea
CSE341: Programming Languages Lecture 12 Equivalence	 Made easier with Abstraction (hiding things) Fewer side effects Not about any "new ways to code something up"
Dan Grossman Autumn 2017	Autumn 2017 CSE341: Programming Languages 2
<section-header><section-header><section-header><list-item><list-item><list-item><list-item><list-item><list-item><table-row></table-row></list-item></list-item></list-item></list-item></list-item></list-item></section-header></section-header></section-header>	<section-header><section-header><section-header><section-header><section-header><section-header><section-header><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></section-header></section-header></section-header></section-header></section-header></section-header></section-header>

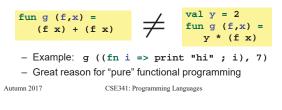
Example

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



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

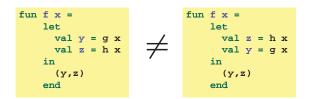
Are equivalent if argument for f has no side-effects



5

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

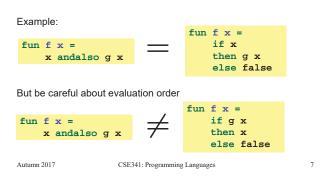


Example: g divides by 0 and h mutates a top-level reference
 Example: g writes to a reference that h reads from



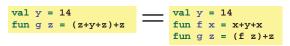
Syntactic sugar

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
- 2. Use a helper function or do not



But notice you need to be careful about environments

=	14 7 = (z+y+z)+z	\neq	val	У	=	7	7+x z)+z
	CSE341: I	Programmin	g Langua	iges			

fung: Autumn 2017

val y

val y

One more

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

let val x = e1in e2 end (fn x => e2) e1

11

- 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 ${\bf x}$ need not be polymorphic, can go from left to right

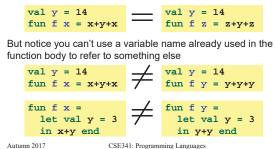
Autumn 2017

CSE341: Programming Languages

Standard equivalences

Three general equivalences that always work for functions - In any (?) decent language

1. Consistently rename bound variables and uses



8

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



What about performance?

According to our definition of equivalence, these two functions are equivalent, but we learned one is awful $\label{eq:equivalence}$

(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	<pre>val y = max xs'</pre>		
else max xs'	in		
	if x > y		
	then x		
	else y		
end			
Autumn 2017 CSE341: Programming Languages			

Different definitions for different jobs

- PL Equivalence (341): given same inputs, same outputs and effects
 - Good: Lets us replace bad \mathtt{max} with good \mathtt{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

Autumn 2017

CSE341: Programming Languages

13