CSE341: Programming Languages
Lecture 12
Equivalence

Zach Tatlock
Winter 2018
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:

\[
\text{fun } f \ x = x + x
\]

\[
\text{val } y = 2
\]

\[
\text{fun } f \ x = y * x
\]

But these next two are not equivalent in general: it depends on what is passed for \( f \)

- Are equivalent if argument for \( f \) has no side-effects

\[
\text{fun } g (f, x) = (f \ x) + (f \ x)
\]

\[
\text{val } y = 2
\]

\[
\text{fun } g (f, x) = y * (f \ x)
\]

- Example: \( g ((fn \ i \Rightarrow \text{print } "hi" ; i), 7) \)
- Great reason for “pure” functional programming
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

\[
\begin{align*}
\text{fun } f \ x &= \text{let} \\
&\quad \text{val } y = g \ x \\
&\quad \text{val } z = h \ x \\
&\quad \text{in} \\
&\quad (y,z) \\
&\text{end} \\
\end{align*}
\]

\[
\begin{align*}
\text{fun } f \ x &= \text{let} \\
&\quad \text{val } z = h \ x \\
&\quad \text{val } y = g \ x \\
&\quad \text{in} \\
&\quad (y,z) \\
&\text{end} \\
\end{align*}
\]

- 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

Example:

\[
\text{fun } f \ x = \begin{cases} 
  x \ \text{andalso} \ g \ x 
\end{cases}
\]

But be careful about evaluation order

\[
\text{fun } f \ x = \begin{cases} 
  x \ \text{andalso} \ g \ x 
\end{cases} 
\neq 
\text{fun } f \ x = \begin{cases} 
  \text{if } g \ x 
  \text{then } x 
  \text{else } \text{false} 
\end{cases}
\]
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

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

val y = 14
fun f y = y+y+y
```

```
fun f x = let val y = 3 in x+y end

fun f y = let val y = 3 in y+y end
```
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
```

```
val y = 14
fun f x = x+y+x
fun g z = (f z)+z
```

But notice you need to be careful about environments

```
val y = 14
val y = 7
fun g z = (z+y+z)+z
```

```
val y = 14
fun f x = x+y+x
val y = 7
fun g z = (f z)+z
```
Standard equivalences

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

3. Unnecessary function wrapping

\[
\begin{align*}
\text{fun } f \ x & = x+x \\
\text{fun } g \ y & = f \ y \\
\text{fun } f \ x & = x+x \\
\text{val } g & = f
\end{align*}
\]

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

\[
\begin{align*}
\text{fun } f \ x & = x+x \\
\text{fun } h \ () & = (\text{print "hi"}; \\
& \quad f) \\
\text{fun } g \ y & = (h()) \ y \\
\text{fun } f \ x & = x+x \\
\text{fun } h \ () & = (\text{print "hi"}; \\
& \quad f) \\
\text{val } g & = (h())
\end{align*}
\]
One more

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

\[
\text{let val } x = e1 \\
\text{in } e2 \text{ end} 
\]

\[(fn \ x \Rightarrow e2) \ e1\]

– These both evaluate \(e1\) to \(v1\), then evaluate \(e2\) in an environment extended to map \(x\) to \(v1\)
– So \textit{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 \(x\) 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)

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
    if x > max xs'
    then x
    else max xs'
```

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
    let
      val y = max xs'
    in
      if x > y
      then x
      else y
    end
```
Different definitions for different jobs

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