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:

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

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

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

- Example: \( g \ ((\text{fn } i => \text{print } "bl" \ ; i), 7) \)
- 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

\[
\text{fun } f \ x = \text{let } \ y = g \ x \text{ in } (y, z) \text{ end} \\
\text{fun } f \ x = \text{let } \ y = g \ x \text{ in } (y, z) \text{ end}
\]

- Example: \( g \) divides by \( 0 \) and \( h \) mutates a top-level reference
- Example: \( g \) writes to a reference that \( h \) reads from
One that really matters

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

\[
\text{map } f \ (\text{map } g \ x) = \text{map } (f \circ g) \ x
\]

Syntactic sugar

Using or not using syntactic sugar is always equivalent

– By definition, else not syntactic sugar

Example:

But be careful about evaluation order

Standard equivalences

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

1. Consistently rename bound variables and uses

\[
\text{val } y = 14 \quad \text{val } y = 14
\]

\[
\text{fun } x = x+y+x \quad \text{fun } x = x+y+z
\]

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

\[
\text{val } x = 14 \quad \text{val } y = 14 \quad \text{val } f = y+y+y
\]

\[
\text{fun } x = x+y+x \quad \text{fun } x = x+y+z
\]

\[
\text{let } \text{val } y = 3 \quad \text{let } \text{val } y = 3
\]

\[
\text{in } \text{x+y and} \quad \text{in } \text{y+y end}
\]

Standard equivalences

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

2. Use a helper function or do not

\[
\text{val } y = 14 \quad \text{val } y = 14
\]

\[
\text{fun } x = x+y+x \quad \text{fun } x = x+y+z
\]

But notice you need to be careful about environments

\[
\text{val } y = 14 \quad \text{val } y = 7 \quad \text{val } g = (x+y+z)
\]

\[
\text{fun } x = x+y+x \quad \text{fun } x = y+y+z \quad \text{fun } g = (f+z)
\]

Standard equivalences

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

3. Unnecessary function wrapping

\[
\text{fun } x = x+y \quad \text{fun } x = x+y
\]

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

\[
\text{fun } x = x+y \quad \text{fun } x = x+y
\]

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

\[
\text{let } \text{val } y = 3 \quad \text{let } \text{val } y = 3
\]

\[
\text{in } \text{x+y and} \quad \text{in } \text{y+y end}
\]

One more

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

\[
\text{let } \text{val } x = \text{val1} \quad \text{fn } x \Rightarrow \text{val2}
\]

– These both evaluate \text{val1} to \text{v1}, then evaluate \text{val2} in an environment extended to map \text{x} to \text{v1}
– So exactly the same evaluation of expressions and result

But in ML, there is a type-system difference:
– \text{x} on the left can have a polymorphic type, but not on the right
– Can always go from right to left
– If \text{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

\[\text{fun max xs = case xs of} \]
\[\text{[ ] => raise Empty} \]
\[\text{[ x:: ] => x} \]
\[\text{[ x::xs' => if x > max xs' then x else max xs']} \]

\[\text{fun max xs = case xs of} \]
\[\text{[ ] => raise Empty} \]
\[\text{[ x:: ] => x} \]
\[\text{[ x::xs' => let} \]
\[\text{val y = max xs'} \]
\[\text{in if x > y then x else y} \]
\[\text{end} \]

Different definitions for different jobs

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