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:

\[
\begin{align*}
\text{fun } f\ x &= x + x \\
\text{fun } g\ (f, x) &= (f\ x) + (f\ x)
\end{align*}
\]

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

\[
\begin{align*}
\text{val } y &= 2 \\
\text{fun } f\ x &= y * x \\
\text{fun } g\ (f, x) &= (f\ x) + (f\ x)
\end{align*}
\]

– Example: \( g\ ((\lambda i. \text{print } "hi" ; i), 7) \)
  – 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

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

– 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:

fun f x = x andalso g x

But be careful about evaluation order

fun f x = x andalso g x

Standard equivalences

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

1. Consistently rename bound variables and uses
But notice you can’t use a variable name already used in the
function body to refer to something else

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
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
– (Actually we studied this before pattern-matching)
Different definitions for different jobs

- **PL 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