CSE341: Programming Languages

Lecture 13
Equivalence;
Parametric Polymorphism

Dan Grossman
Fall 2011
Upcoming schedule

• Today is Wednesday (duh 😊)

• Friday will be an introduction to Racket

• Monday is our midterm, on material up through today
  – Biased toward later lectures because material builds
  – Section will focus on modules and do some review
  – My exams are difficult; possibly a bit harder than samples
    • Don’t panic; it’s fairer that way
  – You can bring one side of one sheet of paper

• Will move into new concepts using Racket very quickly
  – Homework 4 due about a week after midterm and is much more than “getting started with Racket”
Today

1. More careful look at what “two pieces of code are equivalent” means
   – Fundamental software-engineering idea
   – Made easier with (a) abstraction (b) fewer side effects

2. Parametric polymorphism (a.k.a. generic types)
   – Before we stop using a statically typed language
   – See that while generics are very convenient in ML, even ML is more restrictive than it could be
   – (Will contrast with subtyping near end of course)

Won’t learn any “new ways to code something up” today
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:

\[
\text{fun } f\ x = x + x \quad \text{==} \quad \text{val } y = 2 \\
\text{fun } f\ x = y * x
\]

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

- They are if argument for \( f \) has no side-effects

\[
\text{fun } g\ (f,x) = (f\ x) + (f\ x) \quad \neq \quad \text{val } y = 2 \\
\text{fun } g\ (f,x) = y * (f\ x)
\]

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

```plaintext
fun f x = let
  val y = g x
  val z = h x
in
  (y, z)
end
```

- 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
  – Else it’s not actually syntactic sugar

Example:

```
fun f x =
  x andalso g x
```

But be careful about evaluation order

```
fun f x =
  x andalso g x
```

```
fun f x =
  if x
  then g x
  else false
```

```
fun f x =
  if g x
  then x
  else false
```
Standard equivalences

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

1. Consistently rename bound variables and uses

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } f \ x &= x+y+x
\end{align*}
\]

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } f \ z &= z+y+z
\end{align*}
\]

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

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } f \ x &= x+y+x
\end{align*}
\]

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } f \ y &= y+y+y
\end{align*}
\]

\[
\begin{align*}
\text{fun } f \ x &= \\
\text{let } \text{val } y &= 3 \\
in \ x+y \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{fun } f \ y &= \\
\text{let } \text{val } y &= 3 \\
in \ y+y \text{ end}
\end{align*}
\]
Standard equivalences

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

2. Use a helper function or don’t

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } g \ z &= (z+y+z)+z
\end{align*}
\]

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } f \ x &= x+y+x \\
\text{fun } g \ z &= (f \ z)+z
\end{align*}
\]

But notice you need to be careful about environments

\[
\begin{align*}
\text{val } y &= 14 \\
\text{val } y &= 7 \\
\text{fun } g \ z &= (z+y+z)+z
\end{align*}
\]

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } f \ x &= x+y+x \\
\text{val } y &= 7 \\
\text{fun } g \ z &= (f \ z)+z
\end{align*}
\]
Standard equivalences

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

3. Unnecessary function wrapping

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

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

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

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

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

\[
\text{let val } x = e_1 \\
\text{in } e_2 \text{ end}
\]

\[
(fn \ x \Rightarrow \ e_2) \ e_1
\]

- These both evaluate \(e_1\) to \(v_1\), then evaluate \(e_2\) in an environment extended to map \(x\) to \(v_1\)
- 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 \(x\) 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

• CSE341: PL Equivalence: given same inputs, same outputs and effects
  – Good: Let’s us replace bad \texttt{max} with good \texttt{max}
  – Bad: Ignores performance in the extreme

• CSE332: Asymptotic equivalence: Ignore constant factors
  – Good: Focus on the algorithm and efficiency for large inputs
  – Bad: Ignores “four times faster”

• CSE333: 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
Parametric polymorphism

- Parametric polymorphism is a fancy name for “forall types” or “generics”
  - All those 'a 'b things we have leveraged
  - Particularly useful with container types

- Now common in languages with type systems (ML, Haskell, Java, C#, …)
  - Java didn’t have them for many years
  - Will contrast with subtyping near end of course

- Though we have used them, what exactly do they mean…
Example

```ml
fun swap (x, y) = (y, x) (* 'a*'b -> 'b*'a *)
```

Type means “for all types 'a, 'b, function from 'a*'b to 'b*'a”
- Clearly choice of type variable names here doesn’t matter: same type as 'foo*'bar -> 'bar*'foo

In ML the “for all types …” part is implicit, you need not (and cannot) write it out
- Often is explicit in languages

_Fascinating side comment:_ A function of type 'a*'b -> 'b*'a is not necessarily equivalent to `swap` (exceptions, infinite loop, I/O, mutation, …), but if it returns, then it returns what `swap` does (!!)
Instantiation

We can instantiate the type variables to get a less general type

Examples for 'a*''b -> 'b*'a

- int * string -> string * int
- string * string -> string * string
- (int->bool) * int -> int * (int->bool)
- 'a*int -> int*'a
- ...

Non-example

Consider this (silly-but-short) code:

```ml
fun f g = (g 7, g true)
val pair_of_pairs = f (fn x => (x,x))
```

Running this code would work, produce `((7,7), (true, true))`

But `f` will not type-check: type inference fails with conflicting argument types for `g`

`f` does not have type `('a -> 'a * 'a) -> (int * int) * (bool * bool)`
- Body must type-check with *one* type `'a` that callers instantiate

`f` could have type

`(forall 'a, ('a -> 'a * 'a)) -> (int * int) * (bool * bool)`
- Could only be called with a polymorphic function
- But ML has no such type
Why not?

- We just saw that ML cannot type-check a program that makes perfect sense and might even be useful
  - Never tries to misuse any values

- But every sound type system is like that
  - cf. *undecidability* in CSE311
  - Cannot reject exactly the programs that do “hi” (4, 3)

- Designing a type system is about subtle trade-offs
  - Done by specialists
  - Always rejects some reasonable program

- ML preferred convenience of type inference and implicit “for all” “all the way on the outside of types”