CSE 341: Programming Languages

Dan Grossman

Winter 2008

Lecture 11— Equivalence; Syntactic Sugar; Starting Modules

Where are we

- Today: Notions of equivalence; probably start modules
- Monday: Modules; possibly start Scheme
- Wednesday: Scheme basics
- Thursday: Scheme pragmatics; some review
- Friday: midterm
  - Includes modules, but not Scheme
  - You can have one side of one 8.5x11 sheet of paper
  - Old midterms, etc. posted by Monday
  - Will read code, write code, and write English
  - Heavily biased toward later lectures because we have been building
  - (My exams are difficult; don't panic.)

Equivalence

"Equivalence" is a fundamental programming concept

- Code maintenance (simplify code)
- Backward-compatibility (add new optional features)
- Program verification (compare to reference version)
- Program optimization (make faster without breaking it)
- Abstraction and strong interfaces (coming soon!)

But what does it mean for an expression (or program) $e1$ to be "equivalent" to expression $e2$?

First equivalence notion

Context (i.e., "where equivalent")

- Given where $e1$ occurs in a program e, replacing $e1$ with $e2$ does not change e in any way
- At any point in any program, replacing $e1$ with $e2$ makes an equivalent program

The latter (*contextual equivalence*) is much more interesting.

For the former, the body of an unused function body is equivalent to everything (that typechecks).

Second equivalence notion

"how equivalent"

- "partial": $e1$ is equivalent to $e2$ if for any input, any output $e1$ produces is what $e2$ produces
- "total": partial plus $e1$ must terminate if and only if $e2$ terminates

Notice even total contextual equivalence ignores *efficiency* (an exponential algorithm could be "equivalent" to a linear algorithm) according to *this definition*.

Key notion: what is observable?

(For example: Is time-elapsed observable?)

Accounting for "Effects"

Consider whether `fn x => e1` and `fn x => e2` are totally contextually equivalent.

Is this enough? For any environment, `e1` terminates and evaluates to v under the environment if and only if `e2` terminates and evaluates to v under the environment.

We must also consider any *effects* the function may have.

- Mutation, exceptions, printing, modifying files, ...

Functional languages discourage function bodies that do exactly the things that destroy total contextual equivalence.

- For example, *if* you "stay functional" then `(f x) + (f x)` can be replaced by `(f x)*2` *without* consulting what `f` is bound to.
- (Side)-effects are often worth discouraging in any language.

Function equivalences

There are 3 very general things you can do with functions that produce equivalent code. Recognizing them (and their subtle caveats) can make you a better programmer.

1. Systematic renaming of variables
2. "Inlining" by replacing a function call with a body + substitutions
3. Unnecessary function wrapping

Before considering each, it will help to define carefully the notion of *free variables*...

Free variables

An expression e has a set of *free variables*. The definition is:

- For each *use* of a variable, find the *binding* that defines that variable. (This uses the language's *scope rules*.)
- If there is a *use* of x that is in e whose *corresponding binding* is outside e, then x is in the free variables of e.

Example:

```
fun f x =
   let val w = x + y
       val y = fn x => z + y + x
       val q = w + x
   in if g w then x+4 else f (x-1) end
```

Systematic Renaming

Is `fn x => e1` is equivalent to `fn y => e2` where `e2` is `e1` with every `x` replaced by `y`?

(Generally a good property of languages; callers unaffected by code maintenance in callee.)

Scope matters

Is `fn x => e1` is equivalent to `fn y => e2` where `e2` is `e1` with every `x` replaced by `y`?

What if `e1` is `y`?

What if `e1` is `fn x => x`?

Need caveats: `fn x => e1` is equivalent to `fn y => e2` where `e2` is `e1` with every *free* `x` replaced by `y`. But only if `y` is not *free* in `e1`!

Inlining

Is `(fn x => e1) e2` equivalent to `e3` where `e3` is `e1` with every `x` replaced by `e2`?

(Useful for simplifying or specializing code; also a way to think about what a function call is.)

More scope mattering

Is `(fn x => e1) e2` equivalent to `e3` where `e3` is `e1` with every `x` replaced by `e2`?

- Every *free* `x` (of course).
  - Example: `(fn x => (fn x => x)) 17`
- A free variable in `e2` must not be bound at an occurrence of `x`. (Called "capture".)
  - Example: `val y = 4; val z = (fn x => (fn y => x)) y`
- Evaluating `e2` must terminate, not do assignments, not raise exceptions, not print, etc.
  - Because in ML (but not all functional languages), `e2` is evaluated *before* the call
  - Example: `(fn x => x+x) ((print "hi";5))`
- Efficiency? Could be faster or slower. (Why?)

Unnecessary Function Wrapping

A common source of bad style for beginners

Is `e1` equivalent to `fn x => e1 x`?

Sure, provided:

- `e1` effect-free (terminates, no mutation, printing, exceptions, etc.)
- `x` does not occur free in `e1`

Example:

```
List.map (fn x => SOME x) lst
List.map SOME lst
```

Notice variables, constructors, etc. are bound to values, so they are always effect-free (the value is already computed)

Summary so far

We breezed through some core programming-language facts:

- Definition of equivalence depends on observable behavior
- Notion of free variables crucial to understanding function equivalence.
- Three forms of function equivalence:
  - Systematic Renaming
  - Inlining
  - Unnecessary Function Wrapping

Another notion of equivalence we have mentioned but not focused on: *syntactic sugar*

Syntactic Sugar

When all expressions using one construct are totally equivalent to another more primitive construct, we say the former is "syntactic sugar".

- Makes language definition easier
- Makes language implementation easier

Examples:

- `e1 andalso e2` (define as a conditional)
- `if e1 then e2 else e3` (define as a case)
- tuples are really records with field names 1, 2, ...

Note: The error messages used to be even worse because the type-checker worked on a desugared version of your code.

Almost sugar

`#1 e` is not quite sugar because it works for pairs and triples

If we ignore types, then we have this equivalence too:

`let val p = e1 in e2 end` is just `(fn p => e2) e1`.