CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 10— Equivalence and Syntactic Sugar

Where are We

- We have covered enough basics to focus more on concepts now
- You can complete homework 3
- Next Monday will be "Scheme basics"
- This week: Equivalence, modules/abstract-types, parametric polymorphism
- Exam Wednesday April 28...

Exam

- My tests are usually difficult
- You may have 1 side of 1 8.5x11in piece of paper
  - Just a "comfort page"
- Read code, write a little code, write a sentence or two of English
  - Whereas homework was "write code"
  - And you can't "try 50 things you don't understand"
- **Heavily biased toward weeks 3 and 4!**
  - We've been building on earlier material.

Equivalence

"Equivalence" is a fundamental programming concept

- Code maintenance / backward-compatibility
- Program verification
- Program optimization
- Abstraction and strong interfaces

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

First equivalence notion

Context

- Given where $e1$ occurs in a program e, replacing $e1$ with $e2$ makes a program e' equivalent to e
- 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

- "partial": e and e' are equivalent if they input and output the same data (any limits on input?)
- "total": partial plus e and e' have the same termination behavior
- efficiency: e and e' are totally equivalent and one never takes more than (for example) c times longer than the other (or uses much more space or ...)
- syntactic notions: e and e' differ only in whitespace and comments (for example)

Key notion: what is observable? (memory, clock, REP-loop, file-system, ...)

Accounting for "Effects"

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

Is this enough? For all environments E such that `e1` and `e2` typecheck, `e1` terminates and evaluates to v if and only if `e2` terminates and evaluates to v.

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

Purely functional languages have fewer/none, but ML is not purely functional.

In real languages, contextual equivalence usually requires many things.

Nonetheless, "equivalence" to me usually means total contextual equivalence.

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)
- `fun f x y = e` (define with an anonymous function)

More sugar

`#1 e` is just `let val (x,...) = e in x end`

If we ignore types, then we have even more sugar:

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

In fact, if we let every program type-check (or just use one big datatype), then a language with just functions and function application is as powerful as ML or Java (in the Turing Tarpit sense).

This language is called "lambda calculus" – we'll learn a bit more about it later.

Equivalences for Functions

While sugar defines one construct in terms of another, there are also important notions of *meaning-preserving* changes involving functions and bound variables.

They're so important that a goal of language design is that a language supports them.

But the correct definitions are subtle.

First example: systematic renaming

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

Systematic renaming requires care

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` and `y` is not *free* in `e1`.

Note: We can provide a very precise recursive (meta-)definition of *free variables* in an expression.

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

Argument Substitution

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

- Every *free* `x` (of course).
- A free variable in `e2` must not be bound at an occurrence of `x`. (Called "capture".)
  - Always satisfiable by renaming bound variables.
- Evaluating `e2` must have no effects (printing, exceptions, infinite-loop, etc.)
  - Closely tied to the rule that arguments are evaluated to values *before* function application. (Not true for all languages)
  - In ML, many expressions have no such effects (`x`, `#foo x`, ...); much fewer in Java.
- 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` is effect-free
- `x` does not occur free in `e1`

Example:

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

Summary

We breezed through some core programming-language facts and design goals:

- Definition of equivalence depends on observable behavior
- Systematic renaming means context cannot depend on variable names
- Notion of free and bound variables crucial to understanding function equivalence.
- Syntactic sugar "makes a big language smaller" by *defining* constructs in terms of equivalence