

# CSE 341: Programming Languages

Autumn 2005

Lecture 10 — Mutual Recursion, Equivalence, and Syntactic Sugar

# Mutual Recursion

---

You've already seen how multiple functions can recursively call each other in HW 2.

ML uses the keyword `and` to provide different *scope* rules. Example:

```
fun even i = if i=0 then true  else odd  (i-1)
and odd  i = if i=0 then false else even (i-1)
```

Roughly extends the binding form for functions from `fun f1 x1 = e1` to `fun f1 x1 = e1 and f2 x2 = e2 and ... and fn xn = en`.

Syntax gotcha: Easy to forget that you write `and fi xi = ei`, not `and fun fi xi = ei`.

# Mutual Recursion Idioms

---

1. Encode a state machine (see `product_sign` example)
  - Sometimes easier to understand than explicit state values.
2. Process mutually recursive types, as in HW 2

# 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)  $e_1$  to be “equivalent” to expression  $e_2$ ?

# Equivalence I: where?

---

Context (i.e., “where equivalent”)

- 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).

# Equivalence II: how?

---

“how equivalent”

- “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  $\text{fn } x \Rightarrow e_1$  and  $\text{fn } x \Rightarrow e_2$  are totally contextually equivalent.

Is this enough? For all environments,  $e_1$  terminates and evaluates to  $v$  under the environments if and only if  $e_2$  terminates and evaluates to  $v$  under the environment.

Functions produce *values*; may also produce (*side-*) *effects*. Consider both!

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

In real languages, contextual equivalence usually requires many things.

Nonetheless, “equivalence” usually means total contextual equivalence for practical purposes (optimization, reasoning about correctness, etc.).

# Syntactic Sugar

---

“Syntactic sugar causes cancer of the semicolon.”

– Alan Perlis

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  $\text{fn } x \Rightarrow e1$  equivalent to  $\text{fn } y \Rightarrow e2$  where  $e2$  is  $e1$  with every  $x$  replaced by  $y$ ?

## Systematic renaming requires care

---

Is  $\text{fn } x \Rightarrow e1$  equivalent to  $\text{fn } y \Rightarrow e2$  where  $e2$  is  $e1$  with every  $x$  replaced by  $y$ ?

What if  $e1$  is  $y$ ?

What if  $e1$  is  $\text{fn } x \Rightarrow x$ ?

Need caveats:  $\text{fn } x \Rightarrow e1$  is equivalent to  $\text{fn } y \Rightarrow e2$  where  $e2$  is  $e1$  with every *free*  $x$  replaced by  $y$  and  $y$  is not *free* in  $e1$ , and no free  $x$  occurs within the scope of a binding for  $y$  (capture; e.g.:

$\text{fn } x \Rightarrow \text{let } y = 2 \text{ in } x + y \text{ end}$ ; see also next slide.)

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

Next: Is  $(\text{fn } x \Rightarrow e1) e2$  equivalent to  $e3$  where  $e3$  is  $e1$  with every  $x$  replaced by  $e2$ ?

# Argument Substitution

---

Is  $(\text{fn } x \Rightarrow e1) e2$  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
- Syntactic sugar “makes a big language smaller” by *defining* constructs in terms of equivalence
- Notion of free and bound variables crucial to understanding function equivalence.
- Three common forms of function equivalence:
  - Systematic Renaming
  - Argument Substitution
  - Unnecessary Function Wrapping