Immutability
and Referential Transparency
Topics

• Mutation is unnecessary.
• Immutability offers referential transparency.
• Mutation complicates aliasing.
• Broader design considerations
Is immutability an obstacle or a tool?

– Programming experience in 251 and previously
– Readings about language implementation
– Efficiency in space and time
– Reliability
– Maintainability
– Ease of making/avoiding mistakes
– Clarity
– ...

Immutability 3
Mutation is unnecessary.

Patterns for accumulating results without mutation:

– Build recursively
– Create fresh copy with changes
– Explicitly thread state through (e.g., fold):
  • Function does one step, from arguments to result.
  • HOF passes result on to the next step.
Immutability offers *referential transparency*

```
(define (sort-pair p)
  (if (< (car p) (cdr p))
      p
      (cons (cdr p) (car p))))
```

```
(define (sort-pair p)
  (if (< (car p) (cdr p))
      (cons (car p) (cdr p))
      (cons (cdr p) (car p))))
```

Cons cells are immutable. Cannot tell if you copy or alias.
Consider mutation

(define x (mcons 3 4))
(define y (sort-mpair x))

; mutate car of x to hold 5
(set-mcdr! x 5)

(define z (mcdr y))

What is z?
append

(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))

(define x (list 2 4))
(define y (list 5 3 0))
(define z (append x y))
Java security nightmare

class ProtectedResource {
    private Resource theResource = ...;
    private String[] allowedUsers = ...;
    public String[] getAllowedUsers() {
        return allowedUsers;
    }
    public String currentUser() { ... }
    public void useTheResource() {
        for (int i = 0; i < allowedUsers.length; i++) {
            if (currentUser().equals(allowedUsers[i])) {
                ... // access allowed: use it
                return;
            }
        }
        throw new IllegalAccessException();
    }
}
Mutant users!

The problem:
```java
p.getAllowedUsers()[0] = p.current(currentUser);
p.useTheResource();
```

The fix:
```java
public String[] getAllowedUsers() {
    ... return a copy of allowedUsers ...
}
```

Could this happen without mutability?
A biasing on aliasing

Immutability

Aliasing does not affect correctness, just performance.
Other code cannot break your code, regardless of aliasing.
Changing your aliasing cannot break other code.

Document what, not how.

Safe by default, optimize for performance.

Mutability

Aliasing does affect both correctness and performance.
Other code can break your code, depending on your aliasing.
Changing your aliasing can break other code.

Document what and how.

Unsafe by default, optimize for performance and safety.

All the more important for parallelism and concurrency...
Identify dependences between ________.

Racket: immutable natural recursion

\[
\begin{align*}
\text{(define (fib n)} & \text{ (if (< n 2)} \\
& \quad \text{ n} \\
& \quad \text{ (+ (fib (- n 1)) (fib (- n 2)))))
\end{align*}
\]

Racket: immutable tail recursion

\[
\begin{align*}
\text{(define (fib n)} & \text{ (define (fib-tail n fibi fibi+1)} \\
& \quad \text{ (if (= 0 n)} \\
& \quad \quad \text{ fibi} \\
& \quad \quad \text{ (fib-tail (- n 1) fibi+1 (+ fibi fibi+1)})) \\
& \quad \text{ (fib n 0 1))}
\end{align*}
\]

Python: loop iterations with mutation

```
def fib(n):
    fib_i = 0
    fib_i_plus_1 = 1
    for _ in range(n):
        fib_i_prev = fib_i
        fib_i = fib_i_plus_1
        fib_i_plus_1 = fib_i_prev + fib_i_plus_1
    return fib_i
```

And maybe the whole program

Last Week

Recursive calls

Loop iterations

What must we inspect to

And maybe the whole program
A broader PL design theme

Design choices matter. Less can be more (reliable).

Immutability + recursion (vs. mutability + loops) are central:
- Limiting how programs can be expressed
- Making elements more transparent/explicit

This style of design choice often supports:
- Simple reasoning
- Strong default guarantees
- Automated optimization opportunities

It does **not** mean limiting **what** computable functions can be implemented, just **how**.