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
- ...

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-mcar! 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))
```

or

```
x → 2 4
y → 5 3 0
z → 2 4
```

Java security nightmare

```
import java.security.PrivilegedAction;

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
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...

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**.