CS 461

Programming Language Concepts

Gang Tan
Computer Science and Engineering
Penn State University

Imperative Programming

- Oldest and most well-developed paradigm
  - Mirrors computer architecture (von-neumann model)
- Stateful computation
  - A program's state: code, data; both in memory
  - Memory: a map from addresses to values
- Control flow: sequencing, loops, …
- Example Languages
  - Fortran, Pascal
  - C, Ada

Pure Functional Programming

- Program defined as a set of functions
  - Functions are defined in terms of the composition of other functions
- Stateless computation
  - Immutable values; no assignment statements
  - You just construct new values from old values
    - (define x 8); (define x (+ x 1))
    - (append l1 l2) constructs a new list out of old ones
    - Use GC to get rid of unused old values
  - No construct can change the state
- Control flow: no sequencing of statements or loops; use recursion
- Examples: pure Scheme, Core ML, Haskell

Imperative vs. Declarative Constructs

- Imperative constructs
  - int x = 1;
  - x = x +1; // increment x by one
- Declarative constructs (for declaring new entities)
  - (define x 1)
  - (define x (+ x 1))
  - (define (f x) (+ x 1))
- The distinction is between whether
  - changing an existing value (change the state; side effects)
  - or declaring a new value (purity)

  Example:

  ```
  (define x '(1 2 3))
  (define y  (cons 0 x))
  (define x (cdr x))
  ```

  what's the value of y at this point?

What Can Purity Give You? Referential Transparency

- (define (removeDup lst) …)

  ```
  (define lst1 (removeDup '(a b a a a)))
     …. ; some complicated computations here
  (define lst2 (removeDup '(a b a a a)))
  ```
- Observation: no need to recompute lst2
  - removeDup is given the same input, so the outputs should also be the same
- In general, in pure Scheme, calling a function with the same input always produces the same result no matter where the fun call is
  - whenever you refer to it, it is always the same

No Referential Transparency in Imperative Languages

- ```
  int y = 3;
  int f (int x) { return x + y;}
  int main () {
     int z1 = f(3);
        …; // some computation here
     int z2 = f(3);
  }
  ```
- Because the state can change, and the function's return value depends on the state;
  ```

Purity Enables Advanced Techniques

- Such as
  - Memoization; hash consing; automatic parallelization
- Memoization (caching): (fac n) -> (fac_mem n)
  - Initialize an empty association list at the very beginning
  - For (fac_mem n)
    - Check if n has a binding in the assoc list
    - If so, return the value in the binding
    - If not, call (fac n) and add (n, (fac n)) to the assoc list; return (fac n)
  - This is only possible because fac is a pure function

9

Constructs with Side Effects in Scheme

- set!: mutate the value associated with a name

```
(define x 0)
(define (getCounter) x)   ; a zero-argument function
(define (inc) (set! x (+ x 1)))

(inc)
(inc)
(getCounter)  ; returns 2
(inc)
(getCounter)  ; returns 3; note this loses referential transparency
```

- set-car!, set-cdr!
  - (define abcde '(a b c d e))
  - (set-car! abcde 'u) ; the value of abcde afterwards is (u b c d e)
  - (set-cdr! abcde '(d e)) ; the value of abcde afterwards is (u d e)

Constructs with Side Effects in Scheme

- I/O effects
  - (display "hello")
  - file operations
- Sequencing uses the special form begin
  - (begin
    (display "hello")
    (display "world"))
- Iteration
  - do and foreach; see book
- Most of the time, we can program without using these constructs
  - after all, we have done a lot of programming in Scheme already