Algorithm Design and Data Abstraction

CS 146

Brad Lushman

LaTeXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 146 during Winter 2021 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

For any questions, send me an email via https://notes.sibeliusp.com/contact/.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

1

Jan 12

Major theme of CS 146

- side-effect ("impurity")
- programs that *do* things
- imperative programming

General outline

- impure Racket
- C
- low-level machine

Why functional programming first? Why not imperative first?

Imperative programming is harder. Side-effects are not easy things to deal with. For example, text is printed to the screen, keystrokes extracted from the keyboard, values of variables change. All these things change the state of the world. Also, the state of the world affects the program.

If we write a racket program like this one,

```
1    (define (f x) (+ x y))
```

That depends on the value of y. However, if the value of y can change because of the side effects, we have to add a word: it depends on *current value* of y.

Thus the semantics of an imperative program must take into account the current state of the world, even while changing the state of the world.

So there is then a temporal component inherent in analysis of imperative programs. It is not "what does this do?", but "what does this do at this point in time?"

Why study imperative programming at all? It seems it doesn't worth it. "The world is imperative". For example, machines work by mutating memory. Even functional programs are eventually executed imperatively.

. . . "or is it?" Is the world constantly mutating, or is is constantly being reinvented? When a character appears on the screen, does that change the world or create a new one?

Either way, imperative programming matches up with real-world experience, but a functional world view may offer a unique take on side-effects.

Recall from CS 145:

Structural recursion: the structure of the program matches the structure of data.

For example, natural numbers.

```
(define (fact n)                ; A Nat is either
    (if (= n 0) 1               ; 0 or
        (* n (fact (- n 1)))))  ; (+ 1 n) where n is a Nat
```

The cases in the function match the cases in the data definition. The recursive call uses arguments that either stay the same or get one step closer to the base of the data type.

Here is another example on the length of the list.

```
(define (length l)                          ; A (list of X) is empty
    (cond [(empty? l) 0]                    ; or (cons x y) where x
          [else (+ 1 (length (rest l)))]))  ; is an X and y is a (list of X)
```

If the recursion is structural, the structure of the program matches the structure of its correctness by induction.

Claim (length L) produces the length of the list L.

Proof:
Structural induction on L.

Case 1 L is empty. Then (length L) produces 0, which is the length of the empty list.

Case 2 L is `(cons x` L'`)`. Assume that (length L') produces n, which is the length of L'. Then (length L) produces `(+ 1 n)`, which is the length of `(cons x` L'`)`. □

Correctness proof just looks like a restatement of the program itself.

Accumulative recursion one ore more extra parameters that "grow" while the other parameters "shrink".

For example,

```
(define (sum-list L)
    (define (sum-list-help L acc)
        (cond [(empty? L) acc]
              [else (sum-list-help (rest L) (+ (first L) acc))]))
    (sum-list-help L 0))
```

Proof method: induction on an invariant. For example, to prove that `(sum-list L)` sums `L`, suffices to prove `(sum-list-help L 0)` produces the sum of `L`. Let's try to prove by structural induction on `L`.

Case 1 `L` is empty. Then `(sum-list-help L 0)` is `(sum-list-help empty 0)` which gives 0.

Case 2 L = `(cons x` L'`)`. Assume `(sum-list-help` L' `0)` $\Rightarrow$ the sum of L'. Then `(sum-list-help L 0)` is `(sum-list-help (cons x` L'`) 0)` which reduces to `(sum-list-help` L' `(+ x 0))` which is then equal to `(sum-list-help` L' `x)`. Then we are in trouble, because this does not match inductive hypothesis. Proof fails.

So we need a stronger statement about the relationship between `L + acc` that holds throughout the recursion - an invariant.

Proof:
We prove the invariant $\forall$ `L`, $\forall$ `acc (sum-list-help L acc)` produces `acc + (sum-list L)` by structural

induction on `L`.

Case 1 `L` is empty. Then `(sum-list-help L acc)` is `(sum-list-help empty acc)` which gives `acc`, which is equal to the sum of the list + `acc`.

Case 2 `L` is `(cons x` L'`)`. Assume `(sum-list-help` L' `acc)` produces the sum of L' + `acc`. Then `(sum-list-help L acc) = (sum-list-help (cons x` L'`) acc)` $\rightsquigarrow$ `(sum-list-help` L' `(+ x acc))` which is equal to `(sum-list` L'`) + (x + acc) = (+ (sum-list` L'`) x) + acc = (sum-list L) + acc`

Then let `acc = 0`: `(sum-list-help L 0) = (sum-list L)`. □

General recursion: does not follow the structure of the data. Proofs require more creativity.

How do we reason about imperative programs?

Impure Racket:

```
(begin exp_1 ... exp_n)
```

evaluates all of `exp_1, ..., exp_n` in left-to-right order and produces the value of `exp_n`. This is useless in a pure functional setting, but it is useful if `exp_1, ..., exp_(n-1)` are evaluated for their side-effects.

There is an implicit `begin` in the bodies of functions, `lambda`s, `local`, answers of `cond`/`match`. For example,

```
(define (f x)
    ... ; side-effect 1
    ... ; side-effect 2
    ... ; side-effect 3
    ans
)
```

Reasoning about side-effects: for pure functional programming, we have the substitution model, so-called "stepping rules". Can the substitution model be adapted? we can have the "state of the world" an extra input & extra output at each step. So each reduction step transforms the program & also the "state of the world".

How do we model the "state of the world"? For the simple case, it is just a list of definitions. For more complex cases, we need some kind of memory model (RAM) (won't use yet).