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

1.1 Major themes

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.

1.2 Recursion

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 ∀ `L`, ∀ `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)` ⇝ `(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?

1.3 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, `lambdas`, `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).

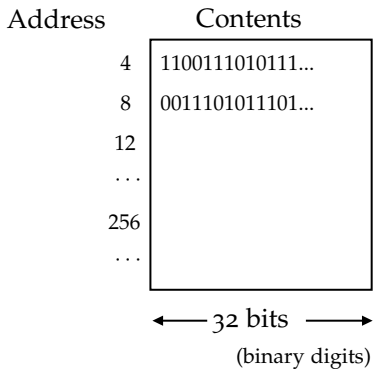
2

Jan 14

2.1 RAM

For now: conceptualization of a RAM (random access machine). Memory is a sequence of "boxes", which are indexed by natural numbers ("addresses"). It contains a fixed size number (say 8 bits or 32 bits). Any box's contains can be fetched $O(1)$ time.

For example, 32-bit RAM:



Will use in a later module, but keep it in mind.

2.2 Modelling Output

It is the simplest kind of side-effect. The "state of the world" here is the sequence of characters that have been printed to the screen. So each step of computation potentially adds characters to this sequence.

> **Note:**
> Every string is just a sequence of characters. Indeed, there is a racket function:
>
> ```
> (string->list "abcd") ===> (list #\a #\b #\c #\d)
> ```

Substitution model $\pi_0 \Rightarrow \pi_1 \Rightarrow \pi_2 \Rightarrow \cdots \Rightarrow \pi_n$ where each π_i is a version of the program obtained by applying one reduction step to π_{i-1}.

In addition to this sequence of programs, now also: $\omega_0 \Rightarrow \omega_1 \Rightarrow \omega_1 \Rightarrow \cdots \Rightarrow \omega_n$ where each ω_i is a version of the output sequence. Because the sequence of characters can only grow, each ω_i is a *prefix* of ω_{i+1} (can't "unprint" characters).

Therefore, we have a combined version: $(\pi_0, \omega_0) \Rightarrow (\pi_1, \omega_1) \Rightarrow \cdots \Rightarrow (\pi_n, \omega_n)$.

Some program reductions will create definitions, (e.g., `local`), and these defined values will eventually change. So let's separate out the sequence of definitions δ.

So we got a triple now: $(\pi_0, \delta_0, \omega_0) \Rightarrow (\pi_1, \delta_1, \omega_1) \Rightarrow \cdots \Rightarrow (\pi_n, \delta_n, \omega_n)$ where δ_0, ω_0, representing the beginning of the program, are empty.

If $\pi_0 =$ `(define id exp) ...`, then we reduce `exp` according to the usual CS 145 (& new CS 146) rules. This may cause characters to be sent to ω. Now `exp` is reduced to `val`. Then remove `(define id val)` from π and add to δ.

If $\pi_0 =$ `exp ...`, then we reduce `exp` by the usual rules, which may cause characters to be sent to ω. Now `exp` is reduced to `val` which is removed from π. So the characters that make up `val` added to ω.

When π is empty, then we are done. So δ, ω is the *state*, that which changes, other than the program itself. ω here is relatively harmless because changes to ω don't affect the running of the program. What about δ? δ is not a problem yet, because variables are not yet changing. All we are doing now is adding new definitions, which is not really a change of state.

How can we affect ω? In Racket, we can do

- `(display x)` which outputs the value of x with no line break
- `(newline)` gives the line break.
- `(printf "The answer is ~a.\n" x)` which is formatted print. The value of `x` replace `~a`. And `\n` is the new line character. As a Racket character on its own: `#\newline`.

In Racket,

```
> (display "Hello")
Hello
> "Hello"
"Hello"
> (begin (display "Hello") 5)
Hello5
> (define x (begin (display "Hello") 5))
Hello
> x
5
```

But then, what do `display`, `newline`, `printf` return? It looks that they don't return anything. We can try following:

```
> (define y (display "Hello"))
Hello
> y          ; y does have a value
> (list y)   ; by a trick
'(#<void>)
```

They return special value `#<void>` which is not displayed in DrRacket. Basically, for functions, that essentially return nothing, and also the result of evaluating (`void`). Functions that return void are called *statements* or *commands* and that's where imperative programming gets its name.

Recall: a Racket function `map`. `(map f (list l1 ... ln))` produces `(list (f l1) ... (f ln))`. It's reasonable to ask what if `f` is a statement? The idea: it is needed for side-effects and produces `#<void>`. Then `(map f (list l1 ... ln))` produces `(list #<void> ... #<void>)` which is not useful.

Instead, now consider `for-each`: `(for-each f (list l1 l2 ... ln))` *performs* `(f l1)`, `(f l2)` ...`(f ln)` and *produces* `#<void>`. For example, we can use it as follows:

```
(define (print-with-spaces lst)
    (for-each (lambda (x) (printf "~a   " x)) lst))
```

This will print out each item in the list with spaces in between and will produce `void` at the end rather than a list of `void`'s. Let's write `for-each`:

```
(define (for-each f lst)
    (cond [(empty? lst) (void)]
          [else (f (first lst)) ; implicit begin
                (for-each f (rest lst))]))
```

or using `if`:

```
(define (for-each f lst)
    (if (empty? lst)
        (void)
        (begin (f (first lst)) (for-each f (rest lst)))))
```

Doing nothing in one case of an `if` condition is common enough that there is a specialized form:

```
(define (for-each f lst)
    (unless (empty? lst) (f (first lst)) (for-each f (rest lst)))) ; implicit begin
```

It evaluates body expressions if the test is false. Similarly, `(when ...)` evaluates body expressions if test is true.

Before we had output, the order of operations didn't matter (assuming no crashes/non-terminations), but now, the order of evaluation may affect the order of output. Also, before we had output, all non-terminating programs could be considered equivalent (not meaningful), but now non-terminating programs can do interesting things (e.g., print the digits of π).

Semantic model should include the possibility of non-terminating programs. What will be the meaning of the non-terminating programs be? It is what the program would produce "in the limit". Here we let Ω to denote the set of possible values of ω, which would include finite & infinite sequences of characters.

But why do we need output? We never used it in CS 145, and Racket has a REPL (Read-Eval-Print-Loop). We can just call functions and see the result. That's what Racket has, but many languages don't have this. Instead, they have compile/link/execute cycle. Under this cycle, the program is translated (by a *compiler*) to a native machine code and then executed from the command line. Then we will only see output if the program prints it. Below is an example of C program.

```
#include <stdio.h>
int main (void) {
    printf("Hello, world!\n");
    return 0;
}
```

Here we have to ask for it if we want something to show up in the screen (line 3).

What about Racket? Here is a use in Racket: tracing program.

```
(define (fact n)
    (printf "fact applied to argument ~a\n" n) ; implicit begin
    (if (= n 0) 1 (* n (fact (- n 1)))))
```

This can aid debugging.

2.3 Modelling input

Let's now talk about the input. We can imagine an infinite sequence consisting of all characters the user will ever press ι. So the model now is $(\pi, \delta, \omega, \iota)$. Every time we need to accept an input character, is the same as removing a character ι.

Here is a small problem: the sequence may *depend* on the output, so the users decide what to input *in response to* what is displayed on the screen. So a more realistic model of input would perhaps not assume all input is available at one.

The alternative: a request for input yields a function consuming one or more characters and producing the next program π, with the user's characters substituted for the read request. For example, a function `(read-line)`, might be modeled as λ `(line) line`. So if user types `"abc"`, as a result of this, we get `"abc"`. Then the entire program reduces to a big "nesting" of input request functions, basically, one function per "prompt". If we supply user input for each prompt, it yields the final result.

Proof techniques for imperative programs will come much later.

2.4 Input in Racket

`(read-line)` produces a string consisting of all characters pressed until the first newline and the string we get does *not* contain the newline.

```
(read-line) ; pops up a little box and lets us to type
Test.
"Test."     ; and get back the string as the result.
> (string->list (read-line)) ; if we type Test.
(list #\T #\e #\s #\t #\.)
```

To read a list of lines, the question then is how do we know when to stop reading? If we look carefully at the box popped up by `(read-line)`, at the end of the box, there is a yellow button, which says "eof" (end of file). When we press that button, it also ends the search for input. "eof" means there is no more input.

```
(define (read-input)
    (define nl (read-line)) ; nl stands for next line
    (cond [(eof-object? nl) empty]
          [else (cons nl (read-input))]))
```

Note that this implementation of `(read-input)` is not tail-recursive.

A more primitive form of input would be `(read-char)` which extracts one character from the input sequence.