2018-01-09 Reasoning about Side-effects

For pure functional programming - substitution model.

Can the substitution model be adapted?

- State of the world is an extra input and output at each step.
- Each reduction step transforms the program transforms the program and also the state of the world.

How do we model "the state of the world"?

- Simple cases: list the definitions.
- More complex cases: memory model - RAM *(which we won't use yet)*

32-bit RAM (Random Access Memory)

- *Address*
- *Content*

Modelling Output

- This is the simplest kind of side-effect
- "State of the world" is the ssequence of characters that have been printed to the screen.
- Each step of computation potentially adds characters to the sequence.
- **Remark**: every string is just a sequence of characters.

Substitution Model

- $\pi_0 \Rightarrow \pi_1 \Rightarrow \pi_2 \Rightarrow \cdots \pi_n$
  - Each π_i is a version of the program obtained by applying one reduction step to π_{i-1}.
- $\omega_0 \Rightarrow \omega_1 \Rightarrow \omega_2 \Rightarrow \cdots \omega_n$
  - Each ω_i is a version of the output sequence.
  - Each ω_i is a prefix of w_{i+1} (can't "unprint" chars).
- Combined: $(\pi_0, \omega_0) \Rightarrow (\pi_1, \omega_1) \Rightarrow (\pi_2, \omega_2) \Rightarrow \cdots \Rightarrow (\pi_n, \omega_n)$
  - But some program reductions create definitions, e.g. local - defined values will eventually change.

- Thus it's better to separate out the sequence of definitions δ_i's:
    - $(\pi_0, \delta_0, \omega_0) \Rightarrow (\pi_1, \delta_1, \omega_1) \Rightarrow (\pi_2, \delta_2, \omega_2) \Rightarrow \cdots \Rightarrow (\pi_n, \delta_n, \omega_n)$, where δ_0, ω_0 empty.

If π_0 = *(define id exp)*...

- reduce exp according to the usual rules (may cause the chars to be sent to ω)
- exp now reduced to a val
- remove *(define id val)* from π and add to δ.

If π_0 = *exp* ...

- Reduce exp by the usual rules (may cause the chars to be sent to ω)
- *exp* now reduced to a value - remove from π.
- Chararacters that make up value are added to ω.
- Stops when π is empty.

δ, ω - the state that which changes, other than the program itself.

- ω as a state is usually harmless - changes to ω don't affect the running of the program.
- δ - not a problem ***yet***, as we haven't introduced changes to variables. The only way to change a variable now is to add new definitions, not really side-effects.

Affecting ω

```
> (display "hello world!")
hello world! # this is a side-effect, not a returned value
> "hello world!"
"hello world!"
> (begin (display "hello") 5)
hello5 # but hello and 5 are "in different colors", ie. different type of outputs.
```

- `(display x)` - outputs the value of x - no line break
- `(newline)` - line break
- `(printf "The answer is ~a.\n" x)` - C-style formatted print, value of *x* replaces *-a*.
    - */n* - newline character (as a Racket char *#\newline*)
- But then, what do *display*, *newline*, *printf* return?
    - They return a special value, ***#<void>***
    - Try `(define y (list (display "hello world")))`
    - This is for functions that essentially return nothing.
- Functions that return void often called "statements", or "commands". (This is where imperative programming gets its name)

Recall: `map`

- What if f is a statement - needed for side-effects - produces *#<void>#*?
- We would get a list of void as return, ie. `(list #<void> #<void> ...)`
- Thus `map` might not be the tool for the job.

Now consider, a new function, a lot like map, but that is suited for the job:

```
(for-each f (list l1 l2 .. ln))
;; performs (f l1), (f l2), ..., (f ln))
;; produces #<void>, which means there is nothing shows up on the screen.
```

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

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

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

To David: Think about the difference between expressions vs. statements; also the use of begin.

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

```
(define (for-each f lst)
    (unless (empty? lst) (f (lrist lst)) (for-each f (rest lst))))
    ;; evaluates body expression if test if false
    ;; simiarly, (when ...) evaluates body expression if test is true
```

Reasoning about output continued…

- **Remark**: the word “output” is equivalent to “side-effect” in this section
- Order
    - Before we had output, order of operations didn’t matter (assuming no crashes/non-termination)
    - But now, order of operation may affect the order of output. Say, printing two things onto the screen, you do care about which one gets printed out first.

- Non-termination
  - Before we had output, all non-terminating programs could be considered equivalent (not meaningful), since you literally don’t get anything back.
  - But now, non-terminating programs can do interesting things! For example, printing the digits of π.
- Thus, Semantic model should include the possibility of non-terminating programs!
  - Since we can’t use the output as the meaning for our program, we need to define the “meaning” of the program in some other ways:
    - what the program would produce **“in the limit”**.
    - Ω (the set of possible values of ω) would include both finite and and infinite sequences of characters.

What if you want to *SAVE* the output?

```
(with-output-to-file
    "inputoutput.txt"                  ;; name of the file
    (lambda () (printf "Text \n")))    ;; "Thunk": a function with no arguments

;; (with-output-to-file) invokes the thunk and stores the output to the file.
;; Better, it doesn't decide what to do with the output - it lets the user decide.
```

Analogy: Linux shell output redirecting (see tutorial).

Why do you need output? (Well, let’s pretend we never used it in CS145)

- Racket has a **REPL** (short for read-evaluate-print loop), so we can just call functions to see the result.
- Many languages don’t operate the same way — they don’t have a REPL — they have a ***compile-link-execute cycle***.
  - Here, the program is translated by the **compiler** to native **machine code**, and then executed from command-line.
  - …which means you only see output if the program prints it.

Hello world in CCCCCCC

```
#include <stdio.h>

int main (void){
    printf("Hello, world! \n");
    return 0;
}
```

A use in racket - tracing programs

```
(define (fact n)
;; (begin(
    (printf "fact applied to argument ~a\n", n)
    (if (zero? n) 1 (* n (fact (sub1 n))))
;; ))

;; We are taking advantage of an implicit begin here.
```

==

Modelling Input

- infinite sequence consisting of all characters the user will ever press.
- ι (iota) - we use this to model the input, and our model becomes $(\pi, \delta, \omega, \iota)$: accepting an input chradcter is to remove a character from ι.