

Invariants and Performance

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 7.6



© Mitchell Wand, 2012-2014

This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Lesson Introduction

- When a function can rely on an invariant, it can be more efficient, because it doesn't need to re-create the information carried by the invariant.
- Many functions are $O(n)$ with a context argument, but $O(n^2)$ without one.
- We'll look at some illustrative examples.

Learning Objectives

- At the end of this lesson the student should be able to
 - show two examples of functions that can be written either with context arguments or without them
 - explain why the version with context arguments are far more efficient
 - explain how context arguments and invariants can lead to better designs.

Example 1: number-list

A `NumberedX` is a `(list Int X)`

A `NumberedListOfX` is a `ListOfNumberedX`

`number-list : ListOfX -> NumberedListOfX`

RETURNS: a list like the original, but with the elements numbered consecutively, starting from 1

```
(number-list (list 22 44 33))  
= (list (list 1 22) (list 2 44) (list 3 33))  
(number-list (list 44 33))  
= (list (list 1 44) (list 2 33))
```

Here's the example we looked at back in Lesson 7.1.

Here was our solution, with a context argument


```
;; number-list-from
;;   : ListOfX Number -> NumberedListOfX
;; GIVEN: a sublist slst
;; WHERE: slst is the n-th sublist of some list lst0
;; RETURNS: a copy of slst numbered according to its
;; position in lst0.
;; STRATEGY: struct decomp on slst : ListOf<X>
(define (number-list-from lst n)
  (cond
    [(empty? lst) empty]
    [else
     (cons
      (list n (first lst))
      (number-list-from (rest lst) (+ n 1)))]))
```

Could we do this directly?

```
(define (number-list lst)
  (cond
    [(empty? lst) empty]
    [else (number-list-combiner
            (first lst)
            (number-list (rest lst)))]))
```

What must **number-list-combiner** do? Let's look at our example.

What must number-list-combiner do?

```
(number-list (list 22 44 33))  
= (number-list-combiner  
  22  
  (number-list (list 44 33)))  
= (number-list-combiner  
  22  
  (list (list 1 44) (list 2 33)))  
=  magic!  
= (list (list 1 22) (list 2 44) (list 3 33))
```

What must number-list-combiner do?

`(number-list-combiner`

`22 (list (list 1 44) (list 2 33)))`
= `(list`
`(list 1 22) (list 2 44) (list 3 33))`

I see a map here

And a cons here

So now we can write the code

```
;; number-list-combiner :
;;   X NumberedListOfX -> NumberedListOfX
;; GIVEN: x1 and ((1 x2) (2 x3) ...),
;; RETURNS: the list ((1 x1) (2 x2) (3 x3) ...)
;; strategy: Use HOFc map on numbered-list
(define (number-list-combiner first-val numbered-list)
  (cons
    (list 1 first-val)
    (map
      ;; (list Number X) -> (list Number X)
      ;; RETURNS: a list like the original,
      ;; but with the first element incremented
      (lambda (elt)
        (list
          (+ 1 (first elt))
          (second elt)))
      numbered-list)))
```

Let's stress-test it...

Let's run both versions on lists of different lengths and see how long they take to run. Code for this is 07-1-number-list-with-stress-tests.rkt.

Times in milliseconds:

length	with context argument	without context argument
1000	0	184
2000	1	506
4000	2	1972
8000	4	8196
16000	7	34907

What do we observe?

- As the length of list doubles,
 - the time with the context argument approximately doubles
 - the time without the context argument approximately quadruples (4x)

What happened here?

- If **lst** has length N , then without an accumulator:
 - **(number-list-combiner n lst)** takes time proportional to N (we say it is $O(N)$)
 - **(number-list lst)** calls **number-list-helper** $O(N)$ times.
 - So the whole thing takes $O(N^2)$.
- The version with accumulator runs in time $O(N)$.
 - much, much faster!

You could do the same thing with Fred-expressions

- Instead of adding new bound variables on the way down, subtract them on the way up:

Function Definition

;; STRATEGY: Use template for FredExp on f

```
(define (free-vars f)
  (cond
    [(var? f) (list (var-name f))]
    [(lam? f) (set-minus
                (free-vars (lam-body f))
                (lam-var f))]
    [(app? f) (set-union
                (free-vars (app-fn f))
                (free-vars (app-arg f)))]))
```

We can write **free-vars** as a straightforward structural decomposition, using the set operations from **sets.rkt**. At each **lam**, we find all the variables in the body, and then remove the lambda-variable from that set. We use **set-union**, rather than **append** or something like it, because we are supposed to return a set.

Fred-expressions

Size	no context arg	with context arg
2559	0	0
81,919	328	47
655,358	2528	390
2,621,439	10732	1591

We saw similar speedups with the FredExp example. From this evidence, it's clear the version with the context argument runs much faster, but there's not enough data here to see whether there's an asymptotic speedup (eg $O(n)$ vs $O(n^2)$)

But performance really isn't the point

- The real point of invariants is to *document the assumptions* that a function makes about the world it lives in.
- Many times, those assumptions are things the function cannot check except with great difficulty
 - e.g., the order contains no duplicates
 - e.g., the inventory is sorted
- You want to check these things once, and then the other functions can rely on them.
- This also means you have a single point of control for these checks
 - this leads to a better design

Summary

- You should now be able to
 - show two examples of functions that can be written either with context arguments or without them
 - explain why the version with context arguments are far more efficient
 - explain how context arguments and invariants can lead to better designs.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Do Problem Set 7.