# On right versus wrong #

*by Stuart Shieber*

We frequently get questions from students of the form “Is this the correct answer to Exercise *n*?” or “Does my solution to Exercise *n* have to have property *p*?” These are natural questions to ask, especially if students are thinking of the exercises as an attempt to get the “right” answer. But CS51 is the course where we move beyond thinking about right and wrong answers in two ways. First is to move to a more nuanced view of alternatives that are better or worse along multiple dimensions, rather than right or wrong – evolving from viewing programming as a *transaction* to viewing it as an *art*. Second is to begin acquiring this nuanced view from inside your person, as opposed to from an oracle (the problem set spec, the instructor, the roommate, the textbook).

In this epistle, I discuss two representative examples, taken from problem set 2.

## Relying on special knowledge

My first example has to do with implicit knowledge about the program. We asked students to write a function `deoptionalize : 'a option list -> 'a list` that extracts values from a list of options. For example, given a list

```
# let l = [None; Some 1; Some 2; None; None; Some 3] ;;
val l : int option list = [None; Some 1; Some 2; None; None; Some 3]
```

we should be able to extract a list of integers:

```
# deoptionalize l ;;
- : int list = [1; 2; 3]
```

A student provides the following code, which generates an inexhaustive match warning:

```
# let deoptionalize (lst: 'a option list) : 'a list =
    List.map (fun x -> match x with Some a -> a)
             (List.filter (fun x -> x <> None) lst) ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: None
val deoptionalize : 'a option list -> 'a list = <fun>
```

and asks "Since `List.filter` already handles the case where `x` is `None`, will I be penalized if I leave the code as is?"

The phrasing in terms of penalties is really a question about whether some alternative implementation is better. And the fact that the question was posed in the first place is a sign that the student already knew the answer. The compiler warning is a signal that something is potentially awry that could cause problems, perhaps not at the moment but down the line as the code evolves. And code always evolves (or at least should be written with an eye towards the possibility).

There are multiple approaches to eliminate the warning. One is to process the recursion directly.

```
let rec deoptionalize (lst: 'a option list) : 'a list =
  match lst with
  | [] -> []
  | None :: tl -> deoptionalize tl
  | Some v :: tl -> v :: deoptionalize tl ;;
```

However, the exercise specifically asked to abstract the recursion using higher-order functions. More importantly, using the map/fold/filter style of higher-order functional programming is natural and elegant for this problem. Why reimplement that implicitly when the abstractions already exist in the `List` library?

If the filter and map approach that the student chose is to be maintained, the fact that the `None` case should never occur can and should be recorded, by raising an exception.

```
let deoptionalize (lst: 'a option list) : 'a list =
  List.map (fun x -> match x with
                     | None -> failwith "deoptionalize: a None snuck in"
```

```
                       | Some a -> a)
             (List.filter (fun x -> x <> None) lst) ;;
```

Reverse application can present this solution in a way that is a bit easier to follow.

```
let deoptionalize (lst: 'a option list) : 'a list =
  lst
  |> List.filter (fun x -> x <> None)
  |> List.map (fun x -> match x with
                        | None -> failwith "deoptionalize: a None snuck in"
                        | Some a -> a) ;;
```

Instead of filtering elements and extracting the contents, an alternative approach is to generate, for each element, a list of values (either zero or one value per list) to be concatenated. Here is an implementation of this approach.

```
let deoptionalize (lst: 'a option list) : 'a list =
  lst
  |> List.map (fun x -> match x with
                        | None -> []
                        | Some v -> [v])
  |> List.concat ;;
```

Another approach is based on accumulating the appropriate values using `fold_right`.

```
let deoptionalize (lst: 'a option list) : 'a list =
  List.fold_right (fun cur accum -> match cur with
                                    | None -> accum
                                    | Some v -> v :: accum)
                  lst [] ;;
```

None of the examples so far are tail-recursive, so if space considerations are important (that is, the function needs to work well on lists of millions of elements), a different approach is needed. Unlike `fold_right`, `fold_left` is tail-recursive, so we might try:

```
let deoptionalize (lst: 'a option list) : 'a list =
    List.fold_left (fun accum cur -> match cur with
                                     | None -> accum
                                     | Some v -> accum @ [v])
                   [] lst ;;
```

This is tail-recursive, but we've gone from a linear time algorithm to a quadratic one (we'll discuss these abstract complexity issues further later in the course), because the repeated calls to append lists (the `@` operator) are traversing ever-longer lists. An alternative is to cons `v` onto the front of the list, generating the reversal of what we want, and then (tail-recursively) reversing the list at the end.

```
  let deoptionalize (lst: 'a option list) : 'a list =
    List.rev (List.fold_left
                (fun accum cur -> match cur with
                                  | None -> accum
                                  | Some v -> v :: accum)
                [] lst) ;;
```

Again, reverse application may make the process clearer.

```
  let deoptionalize (lst: 'a option list) : 'a list =
    lst
    |> List.fold_left (fun accum cur -> match cur with
                                        | None -> accum
                                        | Some v -> v :: accum)
                      []
    |> List.rev ;;
```

Which is the right answer? Some are clearly better than others on all pertinent dimensions. But selecting among the best of this bunch depends on trading off concision, simplicity, transparency, and space and time efficiency. Depending on the context, one or more of these considerations may dominate. The important thing is to understand the large design space even for this simple example, so that these design issues stay in front of mind as programming problems get larger.

## Performance issues

Considerations of performance are worth special consideration in this discussion of right versus wrong. Isn't faster always better? That seems to be the view implicit in the following question from a student about two possible solutions to the problem of concatenating lists of lists, as posed in problem set 2. The student asks,

> For `concat` I know I could do this, which is tail-recursive:
>
> ```
> let concat_tail (lists : 'a list list) : 'a list =
>   let tail_fold_right f l a =
>     List.fold_left (fun x y -> f y x) a (List.rev l)
>   in
>   let rec combine (acc : 'a list) (lst : 'a list) =
>     tail_fold_right (fun a l -> a :: l) acc lst
>   in
>   List.fold_left combine [] lists ;;
> ```
>
> or this, which seems way more expensive but also a lot cleaner :
>
> ```
> let concat_simple (lists : 'a list list) : 'a list =
>   List.fold_left ( @ ) [] lists ;;
> ```
>
> Little decisions like these are the same decisions that I seemed to have points taken off for design in the first pset, and apparently I went with the wrong options for the solutions that I had come up with.

Again, the student intuits the right answer. The second is clearly far superior to the first. It bears on its sleeve how it works, and so is more transparent and maintainable.

But the student is concerned that the simple, elegant, transparent solution will be less efficient, presumably because it applies the append function to longer and longer first arguments. Since append must chase down all of the elements in its first argument, the implementation is worst-case quadratic. (This student is apparently thinking well ahead in the course!)

A simple solution would be to use `fold_right` instead of `fold_left`.

```
let concat_simple_2 (lists : 'a list list) : 'a list =
  List.fold_right ( @ ) lists [] ;;
```

Now the append function's first arguments are just the elements of `lists` in turn, so each element in each sublist is chased down only once. Perhaps the student was scared off by the fact that `fold_right` is not tail-recursive, and thus may run out of stack on long lists, another performance issue. Hence, the student's proposed solution, which attempts to be tail-recursive and linear at the same time.

Manifestly, the `concat_tail` implementation is considerably more complicated than either of the two alternatives ( `concat_simple` and `concat_simple_2` ), and takes a fair effort to convince oneself that it is even correct. Undoubtedly, future readers of the code will spend far longer figuring out what's going on, and will wonder why the simple approach was eschewed.

Here's what legendary computer scientist Donald Knuth says about this issue, which he dubbed "premature optimization":

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

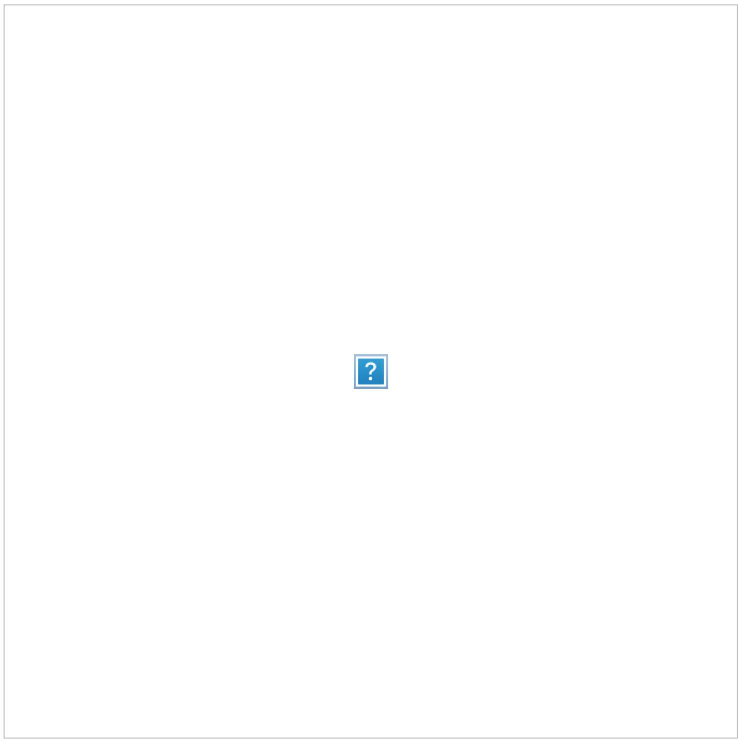
This case of premature optimization is especially unfortunate because the "fast" version `concat_tail` is actually slower than the "slow" version `concat_simple` . The figure below graphs the empirical performance of the functions on lists of lists of increasing length, from 0 to 6,000 total elements.[1] The top line, the "fast" version actually grows superlinearly and is some two to three times slower than the far simpler and more transparent "slow" version.

Of course, it is possible to generate a more efficient version by avoiding the obvious inefficiency of the simple version, namely, the appending of longer and longer constructed lists. Something like the following implementation will help:

```
let concat_tail_2 (lists : 'a list list) : 'a list =
  List.rev lists
  |> List.fold_left (fun tl next -> next @ tl) [] ;;
```

This implementation of `concat` is actually linear and fully tail-recursive.

Its empirical performance is the lower line on the graph in the figure. Nonetheless, the simple versions would still be preferred, except in cases where there is a clear demonstrated need for more efficiency.



**Empirical performance (in seconds on lists of lists of increasing total number of elements) for three implementations of the `concat` function.**

In summary, human time is more important to optimize for than computer time. Simpler, more transparent code is preferable because it will in general optimize human time, unless in the context of the surrounding code it turns out that the inefficiency of the simpler approach is the performance bottleneck in the system in which it participates. In that case, the right solution is to rewrite the code for efficiency, but also to document why you're not using the otherwise obviously clearer alternative.

For this course, *we'll always prefer a more elegant solution over a more efficient solution*, using efficiency considerations just to break ties, unless explicitly stated in the problem specification.

## Conclusion

These by no means exhaust the possible implementations of the two functions I've discussed here. I'm sure I've missed other wholly different alternatives, and would welcome suggestions.[2] But these should be sufficient to make clear that there are multiple approaches that implement a target function specification. There is no right answer, only better or worse ones for different contexts. This is not necessarily an agreeable notion – it would be easier if problems always had a single best answer – but recognizing its truth is central to understanding programming as a creative enterprise, which, after all, is the point of CS51.

1. To hone in on the performance issues as both the number of lists and the length of each list grow, we have both be the square root of the number of elements. (This is not, by the way, the worst case for the `concat_simple` implementation, which would exhibit even worse performance on long lists of short lists.) Times shown are in seconds to execute the call to `concat` 1000 times, timed using the timing functions in the `Absbook` module. ↩

2. For instance, some students wondered about the use of `List.filter_map` in `deoptionalize` to perform the mapping and filtering simultaneously, or the use of the `Option` library functions `Option.get` or `Option.is_some`. ↩