Solving Your Problem by Generalization

CS 5010 Program Design Paradigms “Bootcamp”
Lesson 7.1
Module Introduction

• Some problems are not easily solved using data decomposition as we've defined it. Others are solved only inefficiently by pure data decomposition.

• We introduce two new kinds of generalization to help solve such problems.

• We introduce invariants as a way of recording the assumptions that a function makes about its environment.
Module Outline

• At the end of this module, you should be able to
  – use generalization within a problem to solve the problem
  – use context arguments to generalize over problem contexts
  – write invariants to document the meaning of a context argument
  – explain how invariants divide responsibility between a function and its callers
Lesson Introduction

• In Module 5, we learned about generalizing functions in order to avoid code duplication and establish single points of control.

• In this lesson, we'll extend those techniques to situations where the problem itself demands to be generalized before you can solve it.

• We'll study two examples of this phenomenon.

• This will be a warm-up for the rest of the module.
Learning Objectives for this Lesson

• At the end of this lesson, the student should be able to
  – recognize situations in which it's necessary to generalize a problem in order to solve it
  – write a purpose statement for the generalized problem
  – carry out the rest of the design recipe for such a problem.
Example: mark-depth

(define-struct bintree (left data right)) ;; A BinTree<X> is either
;; -- empty
;; -- (make-bintree BinTree<X> X BinTree<X>)

A Bintree<X> is a binary tree with a value of type X in each of its nodes. For example, you might have BintreeOfSardines. This is, of course, a different notion of binary tree than we saw last week.
Example 2: mark-depth (2)

;; mark-depth : BinTree<X> -> Bintree<Number>
;; RETURNS: a bintree like the original, but
;; with each node labeled by its depth
Here's an example of the argument and result of `mark-depth`. The argument is a `Bintree<String>` and the result is a `Bintree<Number>`, just like the contract says.
Template for BinTree<X>

(define (bintree-fn tree)
  (cond
    [(empty? tree) ...]
    [else (... (bintree-fn (bintree-left tree))
                (bintree-data tree)
                (bintree-fn (bintree-right tree)))]))

If we follow the recipe for writing a template, this is what we get for Bintree<X>.
Filling in the template

(define (mark-depth tree)
  (cond
    [(empty? tree) ...]
    [else (make-bintree
            (mark-depth (bintree-left tree))
            ... (mark-depth (bintree-right tree)))]))

But how do we know the depth?
So again, let’s generalize by adding an accumulator

`mark-depth/a`

: `Bintree<X> Number -> Bintree<Number>`

RETURNS: a bintree like the given one, except that each node is replaced by its depth starting from n

EXAMPLES: see below

STRATEGY: data decomposition on

  `tree : Bintree<X>`
Example (n = 10)
(define (mark-depth/a tree d)
  (cond
    [(empty? tree) empty]
    [else (make-bintree
            (mark-depth/a
             (bintree-left tree) (+ d 1))
            d
            (mark-depth/a
             (bintree-right tree) (+ d 1)))]))

If you are marking tree starting at d, then you should mark its sons starting at d+1
Summary

• Sometimes you need more information than what data decomposition gives you.
• So generalize the problem to include the extra information as an accumulator parameter.
• Design the generalized function.
• Then define your original function in terms of the generalized one.
Lesson Recap

• You should now be able to:
  – recognize situations in which it's necessary to generalize a problem in order to solve it
  – write a purpose statement for the generalized problem
  – carry out the rest of the design recipe for such a problem.