Generative (Non-structural) Recursion

Corky Cartwright
Stephen Wong
Department of Computer Science
Rice University

The Recipe Until Now

- Data analysis and design including generic templates
- For each function in the design
  - Contract, purpose
  - Examples (stated as tests)
  - Template Instantiation
    - Precisely follows the structure of the data processed by the function
    - Using this template, we can do "almost everything"
  - Testing

Structural Recursion

- Is the best problem-solving strategy
  - For the vast majority of functions on recursive data.
  - Yields satisfactory efficiency in most cases.
- Cannot, in principle, compute all computable functions
- Is ill-suited to an important class of problems that technically can be solved using structural recursion but can be solved more cleanly and efficiently using non-structural methods.

Non-structural Functional Programs

- Best explained by presenting some examples before discussing the general template.

  Problem: efficiently sort a list of numbers
  Good solutions: merge-sort, quick-sort

Merge Sort

- Not going to present the actual program because it is an exercise in HW 4.
- Idea:
  - Base case: list of length 0 or 1
  - Inductive case:
    - split the list into two non-empty (almost) equal parts
    - sort each part
  - Merge the two results

- Why non-structural?
  - Base case: list of length 0 or 1
  - Inductive case:
    - split the list into two non-empty (almost) equal parts
    - sort each part (not a substructure!)
    - merge the two results

Quick Sort

- Invented by C.A.R. ("Tony") Hoare
- Functional version is derived from the imperative (destructive) algorithm; less efficient but still works very well
- Idea:
  - Base case: list of length 0 (could be 0 or 1)
  - Inductive case:
    - partition the list into three parts:
      - the singleton list containing first,
      - the list of all items <= first, and
      - the list of all items > first
    - sort the the lists of lesser and greater items
    - return (sorted lesser) || (list first) || (sorted greater)
      where || means list concatenation (append)

Quicksort Breaks Structural Template

```
(define (qsort alon)
  (cond
    [(empty? alon) empty]
    [else
      (local ((define pivot (first alon))
              (define other (rest alon)))
        (append
          (qsort [filter (lambda (x) (<= x pivot)) other])
          (list pivot)
          (qsort [filter (lambda (x) (> x pivot)) other])))]))
```

`qsort` terminates on all inputs. Why?

Not so quick sort

```
(define (qsort alon)
  (cond
    [(empty? alon) empty]
    [else
      (local [(define pivot (first alon))]
        (append
          (qsort [filter (lambda (x) (<= x alon)) other])
          (qsort [filter (lambda (x) (> x alon)) other])))]))
```

This variant may not terminate. Why? Is

`[filter (lambda (x) (<= x pivot)) alon]`

necessarily smaller than `alon`?

A More General Recipe

- Data analysis and design
- Contract, purpose
- Examples
- Template Instantiation (includes header)
  - More flexible than before because non-structural recursion is allowed
- Explicit termination argument if non-structural
- Testing

Generative Template

```
(define (gr-fun problem)  ;; problem is mutliple parameters
  (cond
    [(trivially-solvable? problem)
     (compute-solution problem)]
    [else
     (combine-solutions
       ... problem ...
       (gr-fun (gen-subproblem-1 problem))
       ...
       (gr-fun (gen-subproblem-n problem)))]))
```

where **(*gen-subproblem-1* problem), ..., (*gen-subproblem-n* problem)** are smaller problems of the same form as **problem**.

The Easy Cases

Generalized Structural Recursion

Generalized structural recursion where an input argument **x** is destructured but the subproblems are not immediate components of the **x**. Example: a function that takes a lon and sums consecutive pairs of elements.

```
; add-pairs: lon -> lon
(define (add-pairs lon)
  (cond [(empty? lon) empty]
        [(empty? (rest lon)) lon]
        [else (cons (+ (first lon) (first (rest lon)))
                    (add-pairs (rest (rest lon))))]))
```

This generalized form of structural recursion does not conform to the structural template for lon. It corresponds to strong mathematical induction (also called "course of values" induction) and obviously terminates. The naïve definition of **fib** is another example.

The Easy Cases

Other Forms of Generalized Structural Recursion

- Structural recursion on a tuple such as a pair of numbers. In the standard formalization of arithmetic, corresponds to multiple induction. Some deep theorems in proof theory focus on this issue.
  Example: merge as done in the book.
- Recursion on substructure where the specific substructure is computed by an auxiliary function.
- Example: parse function from next lecture.

Easy Cases cont.

Abstract Structural Recursion

Generative recursion may be structural at an abstract level (if we use a different representation of the argument list). Example: the **`upfrom`** help function from lab 1.

```
; upfrom: nat nat -> list-of nat
(define (upfrom m n)
  (cond [(> m n) empty]
        [else (cons m (upfrom (add1 m) n))]))
```

What does the pair **`(m,n)`** represent? An interval on the number line. Is there another representation we could choose that would make this program structural in the narrow sense used in the book.

Structural Equivalent

This program even has a structural analog when we change to data representation. Let (**`m,k`**) represent the interval from m to **`m+k-1`**, *i.e.*, the set {**`m`**, **`m+1`**, …, **`m+k-1`**}, *i.e*. the interval starting at **`m`** of size **`k`**. Let us call the function using the revised data representation **`Upfrom: nat nat -> list-of nat`**. Then we assert that

```
(upfrom m m+k-1) = (Upfrom m k)
```

Easy Cases cont.

Definition of **`Upfrom`**

```
; Upfrom: nat nat -> list-of nat
(define (Upfrom m k)
  (cond
    [(zero? k) empty]
    [(positive? k) (cons m (Upfrom (add1 m) (sub1 k)))]))
```

This program is structurally recursive and has exactly the same recursive calling structure as **`upfrom`**. Should we use it instead of **`upfrom`** in lab? Book classifies **`upfrom`** as advanced form of structural recursion.

Bonafide generative recursion

- In many cases, generative recursion cannot be interpreted as generalized structural recursion or structural recursion over a different or more abstract representation of the data.
- In this case, we need to ensure that all recursive calls reduce the "size" of some problem *metric* (a function of the argument values). Often this metric is the "size" (expressed as a nat) of the problem inputs. Some common metrics are the length of a list, the depth of a tree, *etc.*
- The simple cases are really simple, e.g., the merge-help function in HW03 if you follow the extra credit path. In that case the sum of the length of both arguments to merge help always decreases.

Sample termination argument

- Quicksort terminates because each recursive call **`(qsort alon)`** reduces the metric **`(length alon)`**. In particular, both **`[filter (lambda (x) (<= x pivot)) other]`** and **`[filter (lambda (x) (> x pivot)) other]`**

  are sublists of **`other`**, which is shorter than **`alon`**
- Without such an argument, a non-structural program must be considered incomplete.

General framework for proving termination

Devise a metric (a size function) for the problem inputs with values of some familiar structural type (often `nat`) and show that each recursive call involves smaller problem inputs than the original one.

In pathological cases, this ordering may require the use of lexicographic ordering on *n*-tuples (or even unbounded sequences like alphabetic words) of data values. These pathologies are *rare* in practice. Not a single occurrence in DrJava code base.

Any *well-founded* partial *ordering* (no *infinite descending chains*) works. The recursive calls must have lesser size according to this ordering. (In structural recursion, the size function is the identity function, and the sizes of recursive calls are *immediately* less the size of the top level call.

We are relying on elementary results in the deep subject or ordinal numbers in set theory.

Why Use Generative Recursion?

- What if we can choose between
  - a structural solution and
  - a generative solution?
- Often, the second is much faster
  - Sorting
  - Simpler example from book: *greatest-common-divisor* (GCD) gcd(6,9)=3, gcd (99, 18) = 9, etc.
    structural version so brain-damaged I could not follow the narrative. I had to infer what the code did.
    Rant: local functions in book often have no contracts!
  - Even better example: searching an ordered list supporting direct access, e.g., an array/vector (but not efficient in functional model if deletions are necessary!)

Are all data types structural?

- Surprisingly delicate question.
- Book says no.
- Walid Taha said no in Comp 210.
- My answer: it depends on how you define the meanings of types. If we use abstract mathematical meanings (where functions are really interpreted as functions) then the answer is no. But if we use the a pedestrian semantics that uses some notion of syntax/code (which is finite!) to represent functions (and other similarly infinite data objects) the answer is yes.
- Which answer do I prefer? As a programmer: “no”; as a language implementor: “yes”. Answer relevant to this class “no”.
- Conceptually I like to think of program data values as abstract mathematical objects. I don't want to think about functions as finite syntactic objects and the correct syntactic representations are more complex than you might think. Moreover, I cannot think of any case where some important property of a program can be established using a pedestrian semantics for functions but not with a mathematical semantics.
- In reasoning about programs, the mathematical point of view is simpler and just as powerful.

Some Algorithm Families

- Sorting and Searching
- Mathematical iteration: bisection, Newton's method.
- Backtracking (traversing a maze, 8 queens)
- Dynamic Programming
- Generally the structural algorithms are so trivial that they typically aren't discussed as *algorithms.* Nothing interesting to say.

Termination Arguments

The details of termination arguments can be tricky and they matter!

Example: binary search

To search for a value v in the ordered sequence S = s1, ..., sn, compare v with the midpoint element sk of S. If equal, done. If < recursively seach s1, ..., sk-1. Otherwise, recursively search sk+1, sn

If we start with an interval N units wide, then we only need a limited number of steps to reach an interval one units wide. In particular, the intervals will proceed as N, N/2, N/4, ..., and will reach size 1 in $\lceil \log_2 N \rceil$ steps. This argument relies on major handwaving. Why? Assumes N is odd so that a well-defined midpoint exists. Assumes N is power of 2 so that N/2, N/4, … make sense, but N cannot be both odd and a power of 2. In practice, the key issue is whether the case N = 2 is handled correctly. If not, binary search can “loop” (infinite recursion).

The Tradeoff (if we can choose)

- How do we chose between
  - a structural solution and
  - a generative solution?
- Speed vs. clarity (brute force) in some cases
- In other cases, there is no *credible* structural algorithm. The structural algorithm may be ridiculously inefficient.
- Chapter 26 has a good example
  - Greatest-common-divisor (GCD)
  - gcd(6,9)=3, gcd (99, 18) = 9, etc.
  - The structural algorithm is worse than crude.
- More typical trade-off: insertion sort vs. quicksort

For Next Class

- Work on HW 4 due next Monday
- Continue Reading:
  - Ch 25-28: Non-structural recursion.
- Start on next homework assignment
  - (mergesort lon) (Problem 26.1.2 but top-down rather than bottom-up version of mergesort)