Halting Measures and Termination Reasoning

CS 5010 Program Design Paradigms

Lesson 8.2

General Recursion is more powerful than structural decomposition

- Functions written using structural decomposition are guaranteed to halt with an answer, but general recursion allows you to write functions that don't always halt.
- So every time we write a function using general recursion, we need to provide some *termination reasoning* that explains why the function really does halt
  - or else warn the user that it may not halt.
  - easiest way to make a termination argument is by supplying a *halting measure*.

Halting Measure (1)

- New required piece of the function header.
- The halting measure is a way of explaining how each of the subproblems are easier than the original
- A halting measure is an integer-valued quantity that can't be less than zero, and which decreases at each recursive call in your function.

Halting Measure (2)

- Since the measure is integer-valued, and it decreases at every recursive call, your function can't make more recursive calls than what the halting measure says.
- In particular, it must halt!

Possible halting measures

- the value of a NonNegInt argument
- the size of an s-expression
- the length of a list
- the number of elements of some set
- a non-negative integer quantity that depends on one of the quantities above

Termination Reasoning

- For each function that uses general recursion you need to give
  - your proposed halting measure
  - some reasoning to show that your proposed halting measure really is a halting measure for your function.

Halting Measure for **decode**

- Proposed halting measure: the size of **sexp**.
- Termination argument:
  - the size of an sexp is always a non-negative integer.
  - If **sexp** is not a number, then **(second sexp)** and **(third sexp)** each have strictly smaller size than **sexp**.
- So **(size sexp)** is a halting measure for **decode**.

There are many ways to define the size of an Sexp. You could, for example, define it as the total number of characters needed to print out the sexp. Can you write this as a function?

Termination reasoning for **`merge-sort`**

- Proposed halting measure: **`(length lst)`**
- Termination reasoning:
  - **`(length lst)`** is always a non-negative integer.
  - At each recursive call, **`(length lst)`** ≥ 2
  - If **`(length lst)`** ≥ 2, then
    `(length (even-elements lst))` and
    `(length (even-elements (rest lst)))`
    are both ***strictly less*** than **`(length lst)`.**
- So **`(length lst)`** is a halting measure for **`merge-sort`**.

Termination Reasoning for **merge**

- Proposed halting measure:
  - **`(length lst1) + (length lst2)`**
- Termination argument:
  - **`(length lst1)`** and **`(length lst2)`** are both always non-negative, so their sum is non-negative.
  - At each recursive call, either **`lst1`** or **`lst2`** becomes shorter, so either way the sum of their lengths is shorter.
- So **`(length lst1) + (length lst2)`** is a halting measure for **merge**.

What do I need to deliver?

- You must write down a halting measure for each function that uses general recursion.
- You don't have to write down the termination reasoning, but you should be prepared to explain it at codewalk.
- If your function does not terminate on some input problems, you should write down a description of the inputs on which your program fails to halt.

A Numeric Example

```
fib : NonNegInt -> NonNegInt
(define (fib n)
  (cond
   [(= n 0) 1]
   [(= n 1) 1]
   [else (+ (fib (- n 1))
            (fib (- n 2)))]))
```

Here's the standard recursive definition of the fibonacci function

A Numeric Example (2)

```
fib : NonNegInt -> NonNegInt
(define (fib n)
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [else (+ (fib (- n 1))
             (fib (- n 2)))]))
```

Let's check to see that the recursive calls obey the contract.

When we get to the recursive calls, if **n** is a NonNegInt, and it is not 0 or 1, then it must be greater than or equal to 2, so **n-1** and **n-2** are both NonNegInt's.

So the recursive calls don't violate the contract.

Termination Reasoning for **fib**

- Proposed halting measure: **n**
- Termination argument
  - **n** is always a non-negative integer (by the contract)
  - At each recursive call, **n-1** and **n-2** are both non-negative integers, and each is strictly smaller than **n.** So **n** decreases at each recursive call.
- So **n** is a halting measure for fib.

What about (fib -1)?

```
(fib -1)
= (+ (fib -2) (fib -3))
= (+ (+ (fib -3) (fib -4))
     (+ (fib -4) (fib -5))
= etc.
```

Oops! This doesn't terminate!

What does this tell us?

- First, it tells us that using general recursion we can write functions that may not terminate.
- We couldn't do this using structural decomposition.
- Is there something wrong with our termination argument?
- No, because the termination argument only says what happens when **n** is a NonNegInt
- -1 is a contract violation, so anything could happen.
- If we want to make the contract Int -> Int , then we need to document the non-termination behavior:

Documenting non-termination

```
fib : Integer -> Integer
Halting Measure:
  If n is non-negative, then n is a
halting measure.
  If n is negative, the function
fails to halt.
```

General Recursion vs. Structural Decomposition

- Structural decomposition is a special case of General Recursion: it's a standard recipe for finding subproblems that are guaranteed to be easier.
  - A field is always smaller than the structure it’s contained in.
- For general recursion, you must always explain in what way the new problems are easier.
- Use structural decomposition when you can, general recursion when you need to.
- Always use the simplest tool that works!

In the definition of function **f** :

`(... (f (rest lst)))` is structural

`(f (... (rest lst)))` is general

You can usually tell just from the function definition whether it is structural or general recursion.

In the first example here, **f** is called on **(rest lst)**, which is a component of the list, and is therefore smaller than **lst**. This is what the list template tells us.

In the second example, **f** is being called some other value that happens to be computed from **(rest lst)**, but that's not the same as **(rest lst)**. So this example is general recursion. There's no telling how big **`(... (rest lst))`** is. If we call f on it, we'd better have a termination argument to ensure that it has a smaller halting measure.

Summary (1)

- We've introduced *general recursion*.
- Solve the problem by combining solutions to easier subproblems.
- Must propose a *halting measure* that documents the "difficulty" of each instance of the problem.
- Must give *termination reasoning* that explains why the proposed halting measure really is a halting measure for this function.
- Structural decomposition is a special case where the data definition guarantees the subproblem is easier.
- Always use the simplest tool that works!

Summary (2)

You should now be able to

- Identify general recursion and distinguish it from structural decomposition.
- Explain the usual structure of a termination reasoning.

Next Steps

- Study the examples of general recursion in 08-1-decode.rkt, 08-2-merge-sort.rkt, and 08-3-fib.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 8.2.
- Go on to the next lesson