Why Recursive Functions Halt

CS 5010 Program Design Paradigms

Lesson 4.6

Introduction

- All of our functions so far always terminated.
- But recursive functions need not terminate!
- In this lesson, we'll study a property that guarantees that a function always halts.
- This property is called "having a halting measure"
- We'll see how to document the halting measure for your function.

Learning Objectives

- At the end of this lesson you should be able to:
  - Identify the halting measure for functions that follow a template
  - Document the halting measure for such functions

Remember **lon-sum**

```
lon-sum : LON -> Number
(define (lon-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (lon-sum (rest lst)))]))
```

Watch this work:

```
(lon-sum (cons 11 (cons 22 (cons 33 empty))))
= (+ 11  (lon-sum (cons 22 (cons 33 empty))))
= (+ 11  (+ 22    (lon-sum (cons 33 empty))))
= (+ 11  (+ 22    (+ 33    (lon-sum empty))))
= (+ 11  (+ 22    (+ 33    0)))
= (+ 11  (+ 22    33))
= (+ 11  55)
= 66
```

Clearly, this function will halt for any LON

- Why?
- Because at every step it works on a shorter and shorter list, so eventually it reaches **empty?** and the function halts.
- In other words, **(length lst)** is a quantity that decreases at every recursive call.

So here's a hypothesis

- If we can find a quantity that decreases at every recursive call to our function, then the function always halts.

Another example: **sum**

```
;; sum :
;;   NonNegInt NonNegInt -> NonNegInt
;; strategy: use template for
;;   NonNegInt on x
(define (sum x y)
 (cond
   [(zero? x) y]
   [else (+ 1 (sum (- x 1) y))]))
```

Example

```
(sum 3 2)
= (+ 1 (sum 2 2))
= (+ 1 (+ 1 (sum 1 2)))
= (+ 1 (+ 1 (+ 1 (sum 0 2))))
= (+ 1 (+ 1 (+ 1 2)))
= 5
```

This one will also work for any non-negative integer x

- At every recursive call, the value of the first argument decreases, so eventually it reaches 0.
- The value of x is a quantity that decreases at every recursive call.
- So this example is consistent with our hypothesis.

Let's look at another example

```
;; foo : NonNegReal -> NonNegInt
(define (foo n)
  (cond
    [(zero? n) 0]
    [else (+ 1 (foo (* n 0.1)))]))
```

This is a silly function, so we won't write out the rest of the purpose statement.

```
(foo 3)
= (+ 1 (foo 0.3))
= (+ 1 (+ 1 (foo 0.03)))
= (+ 1 (+ 1 (+ 1 (foo 0.003))))
= ...
```

Oops! The argument is never equal to 0, so the function never halts.

So we can refine our hypothesis

- If we can find a integer-valued quantity that decreases at every recursive call to our function, then the function always halts.

- All our examples are consistent with this hypothesis.

Let's try another example

```
;; sum2 :
;;   NonNegInt NonNegInt -> NonNegInt
;; strategy: use template for
;;   NonNegInt on x
(define (sum2 x y)
 (cond
   [(zero? x) y]
   [else (+ 2 (sum2 (- x 2) y))]))
```

What if we had used the template incorrectly, and written this program instead?

It still works for even x

```
(sum2 4 3)
= (+ 2 (sum2 2 3))
= (+ 2 (+ 2 (sum2 0 3)))
= (+ 2 (+ 2 3))
= 7
```

But watch what happens when x is odd

```
(sum2 3 3)
= (+ 2 (sum2 1 3))
= (+ 2 (+ 2 (sum2 -1 3)))
= (+ 2 (+ 2 (+ 2 (sum2 -3 3))))
= (+ 2 (+ 2 (+ 2 (+ 2 (sum2 -5 3)))))
= ...
```

Oops! The value of x went negative without being 0. This goes into an infinite loop!

So let's refine our hypothesis again

- Hypothesis: If we can find a non-negative, integer-valued quantity that decreases at every recursive call to our function, then the function always halts.
- This statement is actually true. If the value of our quantity is n, then our function can't possibly recur more than n times: you can't decrease the value of n more than n times without it becoming negative.

Halting Measure

- Definition: a *halting measure* for a particular function is an integer-valued quantity that can't be less than zero, and which decreases at each recursive call in that function.
- This is something you have probably not seen before, so you'll need to pay careful attention.

Examples

- **(length lst)** is a halting measure for **lon-sum**
- the value of **x** is a halting measure for **sum**
- the value of **y** is a halting measure for **prod** (Lesson 4.4).

A function may have more than one halting measure

- The following quantities are halting measures for **sum**:
  - the value of **x**
  - the value of x+4
  - the value of 2*x
- The following quantities are *not* halting measures for **sum**:
  - the value of y
  - the value of -2*x
- But usually there's one "obvious" halting measure, like the ones on the preceding slide.

Don't get confused: "Termination Argument" vs. "Termination Condition"

- The "termination condition" is the condition under which the function halts immediately, eg "the function halts when x reaches 0"
- The "termination argument" is an argument to show that the function always eventually reaches the termination condition.
- The termination argument is your answer to the question: "Why is ⟨the thing you claim is the halting measure⟩ really a halting measure?"

The Halting Measure is a new deliverable

- We will ask you to specify a halting measure for every recursive function you write.
- This is usually easy, eg:

```
HALTING MEASURE: the length of lst
```

or the like.

- When you follow the template, it will almost always be a quantity associated with the template variable.
- The TA may ask you to explain why the thing you called the halting measure really is a halting measure for your function.

Summary

- At the end of this lesson you should be able to:
  - Identify the halting measure for functions that follow a template
  - Document the halting measure for such functions

Next Steps

- Study 04-XXX in the Examples file
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 4.4++
- Go on to the next lesson

GPs: take some from Lesson 8.2, add some for lists.