(Provisional) Lecture 12: Potpourri
10:00 AM, Oct 2, 2017

Contents

1 Introduction 1
2 Operation counting 1
3 Naming new types 3
4 An example: non-empty list types 4
5 List Reversal 5

Announcements

- You are required to attend lab each week, and attend your lab section. If you do not go to the lab you are assigned to, you will not get credit, even if you later go to the correct lab. This is because we have limited resources in terms of computers and TA staff.

Objectives

By the end of this lecture, you will be able to:

- write several kinds of two-argument recursive procedures

1 Introduction

Today’s class is a mix of lots of material. I’ll get to the last part of the lecture Phil did for me, but near the end. Before that, I’ll talk about operation-counting, naming new types, and non-empty list types, and after talking about list-reversal, I’ll also discuss generic types for a little while, if we have time.

2 Operation counting

Recall that I listed earlier all the elementary operations, and we counted that computing the length of the empty list took about 9 operations. If we work through counting operations for a list of length one, it takes about $19 + 9$ operations, which I’ve written that way because in the middle of doing the counting, we end up needing to count the number of operations involves in computing
(\texttt{length empty}), which we already know is 9. So there are 19 new steps, and the 9 we knew about from before.

If we do the same for a list of length 2, we end up doing 19 new steps, and then all the work of computing the length of a list of length one, i.e., it takes $19 + (19 + 9)$ steps in total.

And we start to see a pattern, which we can summarize like this:

Let $L(n)$ be the largest number of steps involved in evaluating (\texttt{length lst}), where \texttt{lst} is any list of length $n$.

Then

\begin{align*}
    L(0) &= 9 \\
    L(n) &= 19 + L(n-1) \quad \text{for } n \geq 1
\end{align*}

The function $L$ is called an operation counting function for the length procedure (or sometimes an “op-count” function, for brevity), and the resulting equation is called a “recurrence equation for $L$”. It’s a set of properties that the function $L$, whatever it may be, must have.

Deriving a recurrence for the op-counting function for a procedure is the first step of analysis of the procedure, so it’s an important skill.

There’s one thing that’s really helpful to know: most of the actual operation-counts in the recurrence will not matter. And it’s best to replace them with constants. So our typical recurrence for the length procedure’s op-counting function would be written instead:

\begin{align*}
    L(0) &= A \\
    L(n) &= B + L(n-1) \quad \text{for } n \geq 1
\end{align*}

The other thing to know is that the second line is almost always (for functions other than \texttt{length}) an inequality, with a less-than-or-equal rather than an equals-sign.

In some procedures, there are multiple base cases, so you may see something like

\begin{align*}
    H(0) &= A \\
    H(1) &= B \\
    H(n) &\leq C + H(n-1) \quad \text{for } n \geq 2
\end{align*}

rather than the simpler form.

There’s a generic set of steps to follow to do this first step of analysis, which we call “deriving the recurrence.”

1. Write “Let $H(n)$ be the largest number of operations involved in evaluating \texttt{myproc} on any list of length $n$”. You may have to vary this slightly for two-argument procedures, but the gist is always the same.

2. Count the number of steps in each base-case. This will almost always lead to an equality of the form

\begin{equation*}
    H(0) = A
\end{equation*}
3. Count the number of steps in the recursive case. Do this in three steps.

(a) Count the number of operations in the recursive case not including those in recursive calls. For example, this might be a constant, or it might depend on \( n \).

(b) Identify the size of the argument to the recursive call. If you have \((\text{my-proc} \ (\text{rest} \ \text{aloi}))\), then this size is \( n - 1 \). But if the recursive call operates on a list consisting of the first, third, fifth, etc. elements of the original list, then this size is more like \( n/2 \); it’s actually \( \lceil n/2 \rceil \), which means “round up \( n/2 \)”.

(c) Write down the work done in the recursive call using your op-counting function. For a single recursive call on the rest of a list, this is \( H(n - 1) \).

(d) Sum up these two op-counts to get the right-hand side of the recursive part of the recurrence relation.

4. Be certain to write the recursive-case recurrence with some “condition” on \( n \), typically “for \( n \geq 1 \)” if the base case was \( n = 0 \), or “for \( n \geq 2 \)” if there were two base cases \( n = 0, 1 \).

For example, a recurrence relation for the procedure \( \text{contains17?} \), which looks like

\[
\text{(define)} \ (\text{contains17?} \ \text{aloi}) \\
(\text{cond}) \\
[[(\text{empty?} \ \text{aloi}) \ \text{false}] \\
[(\text{cons?} \ \text{aloi}) \ (\text{or} \ (= \ 17 \ (\text{first} \ \text{aloi})) \ (\text{contains17?} \ (\text{rest} \ \text{aloi}))))]]
\]

would be: Let \( C(n) \) be the largest number of operations involved in evaluating \( \text{contains17?} \) on any list of length \( n \).

\[
\begin{align*}
C(0) & = A \quad (8) \\
C(n) & \leq B + C(n - 1) \quad \text{for } n \geq 1 \quad (9)
\end{align*}
\]

For the procedure \( \text{foo} \), which looks like

\[
\text{(define)} \ (\text{foo} \ \text{alod}) \\
(\text{cond}) \\
[[(\text{empty?} \ \text{alod}) \ 0] \\
[(\text{empty?} \ (\text{rest} \ \text{alod})) \ 4] \\
[(\text{cons?} \ (\text{rest} \ \text{alod})) \ (+ \ 7 \ (\text{foo} \ (\text{rest} \ (\text{rest} \ \text{alod}))))])]
\]

the recurrence relation would be: Let \( F(n) \) be the largest number of operations involved in evaluating \( \text{foo} \) on any list of length \( n \).

\[
\begin{align*}
F(0) & = A \quad (10) \\
F(1) & = B \quad (11) \\
F(n) & \leq Q + F(n - 2) \quad \text{for } n \geq 2 \quad (12)
\end{align*}
\]
3 Naming new types

Up to now, we’ve restricted domains by putting a condition on an input, as in

;; my-proc: int -> bool
;;
;; input: n, a POSITIVE integer
;; output: ...

But there are cases where (especially when we have lists of lists of lists), the things we’re talking about get pretty complicated, and a good way to simplify is to have names. So if we had to do lots of work with pairs of integers, we might write

;;; Data definitions:
;;; an int-pair is an (int list) of length two
;;;
;;; Examples: int-pair: (list 3 4), (list 5 0)
;;
;;; my-sum: int-pair -> int
;;;
;;; input: p, an int-pair with first item f and second item s
;;; output: the sum of f and s
;;; output: ...

Notice how when I named the input argument, and it had internal parts, I gave them names that could be used in the output description — it’s a useful technique.

For this week’s homework, it’ll be simpler to talk about a “database” than the complex Racket item that is the structure of a database, so naming a new type is appropriate here.

4 An example: non-empty list types

Non-empty lists come up a good deal. For instance, the first procedure operates only on these. We could write

;;; Data definitions:
;;; an int nelist is
;;; an int list of length one
;;; (cons num lst), where num is an int, and lst is an int nelist
;;; nothing else is an int nelist

Notice that in particular, the empty list is not an int nelist.

What’s the template for writing code with these non-empty lists? A good first guess is this

(define (procname alod)
  (cond
    [(empty? (rest alod)) ...]
    [(cons? (rest alod)) ... (first alod) ... (procname ... (rest alod)) ... ]))
but that’s not quite the whole story. Better is this

```
(define (procname alod)
  (cond
    [(empty? (rest alod)) ... (first alod) ... ]
    [(cons? (rest alod)) ... (first alod) ... (procname ... (rest alod)) ... ]))
```

because for a non-empty list, even in the base-case, there’s a `first` that might be involved in the result.

The recurrence for the op-counting function for a procedure that operates on non-empty lists typically starts with \( C(1) = A \) rather than \( C(0) = A \), because the procedure isn’t even defined for lists of length zero!

## 5 List Reversal

To illustrate the value of helpers (and, again, the trick of simplifying a problem by solving a more complicated problem!), let’s reverse a list. You’ll also see that the types of two-argument recursion we defined above are not complete.

We’ll start with a recursive diagram:

- original input: `(list 1 3 4)`
  - recursive input: `(list 3 4)`
  - recursive output: `(list 4 3)`
- original output: `(list 4 3 1)`

It sure seems as if reversing a nonempty list means reversing the `rest` of the list and then adding something on to the `end`, which we’re generally avoiding (for reasons that’ll become evident very soon). As we get down to a zero- or one-element list, reversing gets easy, but the more general case is a mess. For the `empty` list, the answer’s just `empty`. For a list `(list v)`, the result is `(list v)`. But beyond that . . . it’s a mess. Here’s where we use the “bigger helper” trick. Note: we have to call our procedure `my-reverse` because Racket has `reverse` built into it.

We’re going to define `reverse-with-tail`, which takes two arguments, `alod` and `tail` (another list of data), and produces the reverse of `alod` with `tail` appended to it.

```
> (reverse-with-tail (list 1 2 3) (list 4 5))
(list 3 2 1 4 5)
> (reverse-with-tail (list 1 2 3) empty)
(list 3 2 1)
```

It’s pretty clear, if you’ve got `reverse-with-tail`, how to write `my-reverse`:

```
;; Data definition:
;; (a list):
```
;; - empty
;; - (cons `a (`a list))
;
;; my-reverse: (`a list) -> (`a list)
;; input: alod, a list of data
;; output: a new list, consisting of all element of alod, in reverse order

(define (my-reverse alod)
  (reverse-with-tail alod empty))

(check-expect (my-reverse empty) empty)
(check-expect (my-reverse (list 1 2)) (list 2 1))
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

So how do we write reverse-with-tail? The answer is not to just write reverse and then call append (a built-in that you’re writing for this week’s HW!), because we’re using this procedure to write reverse!

Let’s look at some recursive cases. To do so, we have to decide how we’re going to recur. Let’s try this example.

- original input: (list 1 2) (list 17 18)
  - recursive input: (list 2) (list 17 18)
  - recursive output: (list 2 17 18)

- original output: (list 2 1 17 18)

That looks horrible. We need to perform surgery to stick that “1” in there. How about this:

- original input: (list 1 2) (list 17 18)
  - recursive input: (list 2) (list 1 17 18)
  - recursive output: (list 2 1 17 18)

- original output: (list 2 1 17 18)

Wow! By changing both lists— one getting simpler, the other getting more complicated — we’ve got ourselves a solution!
(define (reverse-with-tail alod start)
  (cond
    [(empty? alod) start]
    [(cons? alod) (reverse-with-tail (rest alod)
       (cons (first alod) start))])))

(check-expect (reverse-with-tail empty empty) empty)
(check-expect (reverse-with-tail (list 1 2) empty) (list 2 1))
(check-expect (reverse-with-tail (list 1 2 3) (list 4 5)) (list 3 2 1 4 5))

Done! Woo hoo! The overall result is

;; Data definition:
;; (`a list):
;; - empty
;; - (cons `a (`a list))
;; reverse-with-tail: (`a list) (`a list)-> (`a list)
;; input: alod, a list of data
;; start, another list of data of the same type as alod
;; output: a new list, consisting of all element of alod, in reverse order,
;; followed by those of start, in their original order.

(define (reverse-with-tail alod start)
  (cond
    [(empty? alod) start]
    [(cons? alod) (reverse-with-tail (rest alod)
       (cons (first alod) start))])))

;; my-reverse: (`a list)-> (`a list)
;; input: alod, a list of data
;; output: a new list, consisting of all element of alod, in reverse order

(define (my-reverse alod)
  (reverse-with-tail alod empty))

(check-expect (my-reverse empty) empty)
(check-expect (my-reverse (list 1 2)) (list 2 1))
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

Now you might ask yourself a couple of questions here:

- I see that it solves the problem, but what on earth motivated you to write reverse-with-tail, Spike?
- How many arguments can a helper-procedure like this have? Could it have 3, or 4, or twelve?

The answer to the second question is pretty much “two or three”. There’s a tendency to use these things in a way that’s sloppy and that makes it hard to express what your procedure actually does. If you can’t write a crisp specification for the helper, you’re probably doing something wrong.
As for the first question, the answer is that in looking at the recursive diagram, I saw that right from the start (unlike in the \textit{length} procedure), I already had part of the answer. In reversing \((\text{list 1 2 3})\), the first of the input list was in fact, something that needed to be right at the start of building up an answer. That meant that I wanted to have a gradually accumulating answer, and the “tail” was a way to keep track of that as I went deeper and deeper into the recursion.

Perhaps a better name for our helper would be \textit{reverse-with-partial-result}.

Notice that in this two-argument recursion, while one argument shrinks, the other is growing at every step of the recursion. That’s in contrast to \textit{shorter-list}, where both grew smaller, and in contrast to \textit{intersect} where one argument shrinks and one argument stays the same size.

\begin{itemize}
\item \textbf{Question}: Are there any other possibilities, like one argument staying the same and the other getting more complicated, or both getting more complicated?
\item \textbf{Answer}: In those cases, repeated recursive invocations of the procedure process more and more complicated data, and never terminate: the (\texttt{empty? lst}) clause never gets evaluated, and the other clause always ends up making a recursive call. (There’s one more case: both arguments stay the same size; that doesn’t work either.) It’s \textit{essential}, in writing recursive procedures in CS17, that at least one argument gets simpler in each call, or proving that the procedure terminates becomes infeasible.
\end{itemize}