Lecture 13: Analysis, New Type-Names
10:00 AM, Oct 2, 2019

Contents

1 Review 1
2 Solving a Recurrence 1
3 More Plug and Chug 2
4 Generic Types 4
5 Summary 5

1 Review

Last time we looked at our final version of the reverse-with-tail procedure (see lecture 12 notes for the full story).

\[
\text{(define (reverse-with-tail aloi tail)}
\begin{cases}
\text{[(empty? aloi) tail]}
\text{[(cons? aloi) (reverse-with-tail (rest aloi) )]
\text{ (cons (first aloi) tail)))}}
\end{cases}
\]

We have,

“Let \( H(n) \) be the number of elementary operations done in evaluating (reverse-with-tail aloi tail) on any input where aloi is a list of length \( n \).”

Notice that this procedure does not involve any decision making using if statements that determines the number of operations. Therefore, we can find the exact number of operations instead of the upper bound!

Then we showed that \( L \) satisfied a pair of equations we called a recurrence relation, namely

\[
\begin{align*}
H(0) &= A \quad (1) \\
H(n) &= B + H(n-1), \text{ for } n > 0. \quad (2)
\end{align*}
\]

We have seen the same recurrence relation for length, contains7 and reverse-helper as well! Now let us see how to solve this recurrence.
2 Solving a Recurrence

Step 1 is writing down the recurrence relation.

\[ Q(0) = A \]  \hspace{1cm} (3)
\[ Q(n) = B + Q(n-1), \text{ for } n > 0. \]  \hspace{1cm} (4)

Observe that we have a value for \( Q(n) \) in terms of \( Q(n-1) \). This is a recursive description of \( Q \).

To solve a recurrence relation is to find a non-recursive, closed form formula for \( Q(n) \), that does not involve \( Q(n-1) \) or any other previous values. In special cases, as we’ve seen so far, we’ll be able to write, 
\[ Q(n) = \ldots \]
But more often we’re at most able to say, 
\[ Q(n) \leq \ldots \]

The technique we will be using to solve recurrences is called **Plug and Chug**. Here’s how it works.

We can start to figure out a little more about the function \( Q \) by evaluating a few cases using the recurrence. Because the exact operation counts don’t matter for our application, we replace them with letters.

For **Step 2**, we’ll try to find a pattern by computing \( Q(0), Q(1), Q(2), Q(3) \):

\[ Q(0) = A \]  \hspace{1cm} (5)
\[ Q(1) = B + Q(0) = B + A \]  \hspace{1cm} (6)
\[ Q(2) = B + Q(1) = B + B + A \]  \hspace{1cm} (7)
\[ Q(3) = B + Q(2) = B + B + B + A \]  \hspace{1cm} (8)
\[ \ldots \]  \hspace{1cm} (9)
\[ Q(n) = nB + A \text{ for all natural numbers } k \]  \hspace{1cm} (10)

where that last step was an example of inductive reasoning, i.e., “noticing a pattern”.

It’s tempting to simplify the algebra early on, but as patterns get more complicated, it’s hard to know what terms to combine.

**Step 3** is to prove that our conjecture is correct. This step is fairly mechanical as all these proofs follow the same logic. We will be learning how to prove our conjectures in the coming classes.

3 More Plug and Chug

For the following recurrence relations, try using plug and chug to solve them,

1. \[ Q(0) = A \]
   \[ Q(n) = Q(n-1), n > 0 \]

\[ Q(0) = A \]
\[ Q(1) = Q(0) = A \]
\[ Q(2) = Q(1) = A \]
\[
\ldots
\]
\[ Q(n) = A \]

This function takes a constant number of operations to run.

2. \( Q(0) = A \)
\[
Q(n) = 2Q(n-1), n > 0
\]
\[
Q(0) = A
Q(1) = 2 \times Q(0) = 2 \times A
Q(2) = 2 \times Q(1) = 2 \times 2 \times A
Q(3) = 2 \times Q(2) = 2 \times 2 \times 2 \times A
\ldots
Q(n) = 2^n A
\]

The number of operations this function takes is growing exponentially as \( n \) increases. Notice how a mere factor of 2 in the recurrence relation made such a large difference.

3. \( Q(0) = A \)
\[
Q(n) = 2Q\left(\frac{n}{2}\right), n > 1
\]
Apply this only to powers of 2, \( n = 2^k \)
\[
Q(1) = A
Q(2) = 2 \times Q(1) = 2 \times A
Q(4) = 2 \times Q(2) = 2 \times 2 \times A
Q(8) = 2 \times Q(4) = 2 \times 2 \times 2 \times A
\ldots
Q(n) = nA, \text{ where } n \text{ is a power of } 2.
\]

The number of operations this function takes is grows linearly.

4. A favorite: \( Q(0) = A \)
\[
Q(n) = B + C(n) + Q(n-1), n > 0
\]
\[
Q(0) = A
Q(1) = B + C \times 1 + Q(0) = B + C \times 1 + A
Q(2) = B + C \times 2 + Q(1) = B + C \times 2 + B + C \times 1 + A = 2B + (1 + 2)C + A
Q(3) = B + C \times 3 + Q(2) = B + C \times 3B + C \times 2 + B + C \times 1 + A = 3B + (1 + 2 + 3)C + A
\ldots
Q(n) = nB + (1 + 2 + 3 + \ldots + n)C + A
\]

This is a great example for why we should wait to simplify our algebra until after we notice a pattern. Had we written our plug and chug as below, the pattern would not have immediately jumped out.
\[
Q(0) = A
Q(1) = B + C \times 1 + Q(0) = B + C + A
\]
\[ Q(2) = B + C \times 2 + Q(1) = 2B + 3C + A \]
\[ Q(3) = B + C \times 3 + Q(2) = 3B + 6C + A \]

Going back to our last conjecture, notice that it is still not in closed form as it has a term that is 
\((1 + 2 + 3 + \ldots + n)\). Luckily, there is a very nice closed form formula for that summation! It is 
called Gauss’s formula. Here is a basic derivation:

\[ S = 1 + 2 + 3 + \ldots + (n - 1) + n \quad \text{...the sum we want to compute} \]  \(11\)
\[ 2S = 1 + 2 + 3 + \ldots + (n - 1) + n + (1 + 2 + 3 + \ldots + (n - 1) + n) \quad \text{...multiply both sides by 2} \]  \(12\)
\[ 2S = n + (n - 1) + \ldots + 3 + 2 + 1 + 1 + 2 + 3 + \ldots + (n - 1) + n \quad \text{...rearrange the terms} \]  \(13\)
\[ 2S = (n + 1) + (n - 1 + 2) + \ldots + (2 + n - 1) + (1 + n) \quad \text{...add vertical terms} \]  \(14\)
\[ 2S = n(n + 1) \]  \(15\)
\[ S = \frac{n(n + 1)}{2} \]  \(16\)

Using Gauss’s formula, our last conjecture will become,

\[ Q(n) = nB + \frac{n(n + 1)}{2}C + A \]

And we have a closed form!

The formula for sums of squares is:

\[ 1 + 4 + 9 + \ldots + n^2 = \frac{n(n + 1)(2n + 1)}{6} \]

There are also similar formulas for sum of cubes, 4th powers, etc. but we won’t be needing them. It
is interesting to note some patterns in these formulas, for example, the first term for the sum of \(n^k\)s
is always \(\frac{k+1}{k+1}\).

### 4 Generic Types

Let’s take a little break from analysis and look at type-signatures again. We’ve written type-sigs for
all the procedures we’ve created, but what about the built-ins? What the signature for \texttt{cons}? Well,
\texttt{cons} can be used to build up int lists, or string lists, or bool lists (or even lists that contain ints
and bools and strings, but we don’t allow that in CS17, so we’re going to pretend that Racket’s
\texttt{cons} doesn’t either).

We’d like to write

\[
\begin{align*}
\&; \texttt{cons: int } \star \texttt{ (int list) } \rightarrow \texttt{ (int list)} \\
\&; \texttt{cons: string } \star \texttt{ (string list) } \rightarrow \texttt{ (string list)} \\
\&; \ldots
\end{align*}
\]
but since the number of possible examples is infinite, that’d take a long time. Racket documentation 
would write this type signature with something like,

```scheme
cons: any * list -> list
```

However, in CS17 we will use generic types. So we’re going to say that cons takes a thing of type \( \alpha \), 
and a list of items of type \( \alpha \), and produces a new \( \alpha \)-list. The name \( \alpha \) here is kind of a placeholder: 
it can mean ‘int’, or ‘bool’, or ‘((bool list) list)’ or almost anything else. Our type signature would 
look like,

```scheme
;; cons : \( \alpha \) * (\( \alpha \) list) -> (\( \alpha \) list)
```

Technical papers use greek letters like this for “type names”, but due to historial practice, programs 
are almost always written in character sets that don’t contain greek letters, so we (or more precisely, 
the authors of OCaml, whom we’re imitating) substitute ‘\( a \) for \( \alpha \).

So now we can say

```scheme
;; cons : 'a * ('a list) -> ('a list)
```

Some other examples are,

```scheme
;; member? : 'a * ('b list) -> bool [for us 'a * ('a list) -> bool instead]
;; string? : 'a -> bool
```

Yay!

While we’re at it, let’s look at

```scheme
(define (apply f a) (f a))
```

What’s the type-signature for apply? Well, \( f \) could be `not` and \( a \) could be boolean. Or \( f \) could be 
`first`, and \( a \) could be some list. Essentially, \( f \) should be a procedure and \( a \) should be an argument 
to \( f \).

The type signature we want is this:

```scheme
;; apply : ('a -> 'b) * 'a -> 'b
```

I’ve introduced a new “type” here, an ‘arrow type’; it’s what we use to describe the type of a 
`function` rather than atomic or list types. Note that ‘\( a \) and ‘\( b \) might denote different types, or they 
might both be, say, `int` (with \( f \) being `succ`, for instance).

## 5 Summary

### Ideas

- When we want to analyze a procedure that operates on inputs of various sizes, we typically 
call the size of the input \( n \), and say “Let \( P(n) \) be the largest number of operations performed 
by the procedure on any input of size \( n \).”
- When performing runtime analysis, we ignore constants.
Skills

- We know how to use “Plug-n-Chug” to generate an equation that will tell us something about the maximum number of operations a procedure will take on an input of size $n$. In later lectures, we will use this equation to prove something about the run time of our procedure as compared to other procedures.

- How to use generic types in writing type signatures.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)