

Contraction and Scan

1 Contraction

Recall that *divide and conquer* is a technique for solving a problem by reducing it to *multiple* smaller versions of itself, solving those recursively, then combining the solutions to the smaller problems to solve the bigger problem. Divide-and-conquer is highly attractive for parallel algorithms, because the recursive calls can usually be done in parallel. However, divide-and-conquer algorithms tend to do a lot of work if the combine step is not constant time.

Contraction is a similar technique to divide-and-conquer that can sometimes achieve better cost bounds. Instead of breaking a problem into multiple smaller versions of itself, contraction *shrinks* a problem into **one** smaller version of itself, solves that recursively, then uses that solution to figure out the solution to the original (bigger) problem.

Definition: Contraction-based algorithm

A contraction algorithm solves a problem by **contracting** to generate **one** smaller instance, solves that instance **recursively**, then **expanding** to solve the original problem.

We can contrast this definition with divide and conquer to see its similarities and differences:

- **Base Case:** When the problem size is sufficiently small, solve it directly (without further recursion). (*same as divide and conquer*)
- **Inductive Step:**
 1. **Divide Contract** the instance into ~~many~~ one smaller instances,
 2. **Recursively** solve the smaller instances,
 3. **Combine Expand** the solutions to solve the original instance.

Because it only performs one recursive call instead of many, contraction can sometimes perform less work than divide and conquer on the same problem. As a warm up, let's solve the familiar *reduce* problem, which we already solved with divide-and-conquer, using contraction.

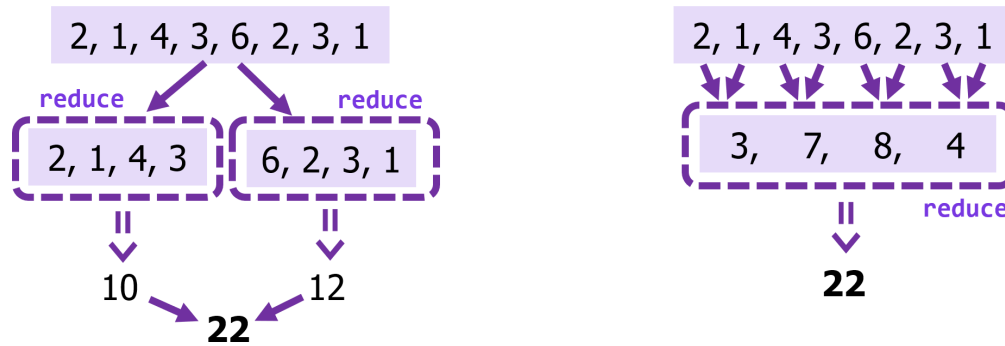
1.1 Reduce via Contraction

The idea of contraction is to shrink a problem to a smaller problem that still contains enough information to help us solve the bigger problem. Given a sequence of n elements that we want to reduce, can you think of a sequence of $n/2$ elements that has the same answer?

Here's a simple idea: pair up adjacent elements and reduce them (add them together). This shrinks the sequence to one that is half as big and has the same total answer as the original.

Now we just recursively reduce this smaller sequence and we are done.

This figure shows the difference between the divide-and-conquer algorithm for reduce (left) and the contraction algorithm for reduce (right).



Algorithm: Reduce via Contraction

```

fun reduce(f : (T, T) -> T, I : T, S : sequence<T>) -> T:
  match length(S) with:
  case 0: return I
  case 1: return S[0]
  case _:
    B = parallel [f(S[2*i], S[2*i+1]) for i in 0...|S|/2-1]
      + [S[|S|-1]] if |S|%2 == 1 else []
    return reduce(f, I, B)

```

To contrast the two techniques, note that in the divide-and-conquer approach, the divide step and the combine step both have $O(1)$ cost, while in the contraction-based approach, the contraction step costs $O(n)$ work and $O(1)$ span, and the expansion step costs $O(1)$. Initially, this might seem more expensive than divide and conquer, but the savings come from the fact that the algorithm only needs to make **one** recursive call.

An important implication of this is that since there is only one recursive call, we no longer get any parallelism from the recursion. Instead, the contraction (the step which combines pairs of elements) and expansion steps (trivial for reduce) must provide the parallelism.

Theorem: Cost of Contraction-Based Reduce

Assuming that f can be evaluated in constant time, reduce implemented with contraction also costs $O(|S|)$ work and $O(\log |S|)$ span.

Proof. Let $n = |S|$. At the root, the contraction step performs $O(n)$ work. Each recursive call halves the size of the sequence, so for the work recurrence we have

$$W(n) = W\left(\frac{n}{2}\right) + O(n).$$

Notice the differences between the work recurrence of the divide-and-conquer algorithm: We

no longer have a 2 factor in front of the $W(n/2)$ because there is only one recursive call, and the local work is now $O(n)$ rather than $O(1)$ due to the contraction step.

We can solve this using the brick method. The local work at the root is n , and the work at the (only) child is $n/2$, which means that it is **root dominated**. Therefore the work is $O(n)$. This is another potential advantage of contraction; since there is only one recursive call, the recurrence relation is often much easier to solve!

Similarly, the span at the each level is $O(1)$ since the contraction step is just a tabulate and the expansion step is trivial, so the span recurrence is

$$S(n) = S\left(\frac{n}{2}\right) + O(1).$$

This recurrence is **balanced** since every level has the same $O(1)$ local cost. The recurrence has $O(\log n)$ levels so the span solves to $O(\log n)$. $\square$

2 Scan

In the last lecture, we studied the **reduce** problem and its applications to parallel algorithm design. We saw that reduce is very useful since we can apply a wide range of *associative* combining functions to it to compute various aggregates like sums, products, minimums, maximums, etc.

Now, we study the **scan** problem, which generalizes reduce to compute not just the total, but the reduction of *every prefix* of a sequence with respect to a given associative function. With scan, we can solve even more interesting problems.

Definition: Scan

Given a sequence S , an associative function and an identity, scan computes the exclusive prefix sums and the total sum of S with respect to the associative function and identity.

Note that scan actually returns *two* objects. The first is a sequence containing the *exclusive* prefix sums of S (meaning that at position i , the sum consists of all elements that occur *strictly before*, not including, position i). The second is the total sum since it is not included as part of the prefix sums, since they are exclusive.

```
scan(plus, 0, [2, 1, 4, 3, 6, 2, 3, 1])
-> ([0, 2, 3, 7, 10, 16, 18, 21], 22)
```

To make the semantics concrete, the output of scan is equivalent to the following brute-force implementation. Of course we will aim to implement something much more efficient.

```
fun scan(f : (T, T) -> T, I : T, S : sequence<T>) -> (sequence<T>, T):
  return tabulate(fn i => reduce(f, I, subseq(S, 0, i)), |S|, reduce(f, I, S))
```

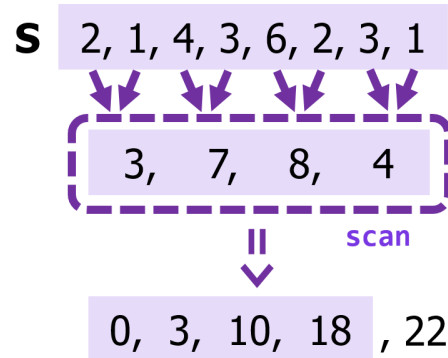
Remark: Scan via Divide-and-Conquer

Scan can be implemented via divide-and-conquer with a cost of $O(n \log n)$ work and $O(\log n)$ span. Try to write it. This is faster than brute force, but we can still do better.

2.1 Scan via Contraction

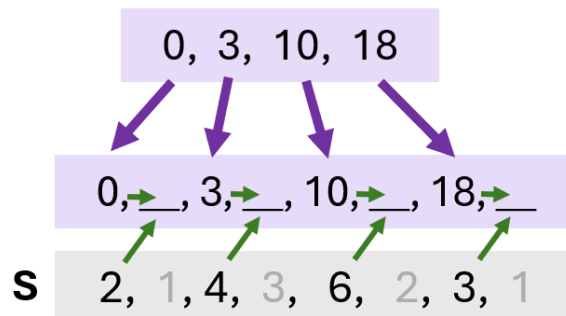
The implementation of scan via contraction starts with the same steps as reduce. Given a sequence S , we contract it by pairing up adjacent elements and combining the pairs with the combining function f . We then recursively scan the contracted sequence.

The tricky part is that the result of the recursive call is no longer the entire answer to the problem like it was for reduce. Instead, we need a non-trivial **expansion** step, where we use the result of the recursive call to figure out the solution to the original problem. To figure this out, let's look at an example of what the recursive call returns and see what information it offers us.



The recursive call gave us back the sequence $[0, 3, 10, 18]$ and the total value. So, what useful information does it contain? Clearly the total is useful, but what about the prefix sums? Remember that the sequence we need to produce is $[0, 2, 3, 7, 10, 16, 18, 21]$. The recursive result contains the **even-indexed elements** of the answer!

Therefore, the expansion step just has to fill in the odd-indexed elements. To do so, we observe that an element at index i is just the sum of the previous prefix sum and the element at position $i - 1$. Critically, none of them depend on the earlier ones, so they can all be filled in in parallel!



This gives us the following efficient parallel implementation of scan. It is almost identical to the contraction-based implementation of reduce, except that it returns two objects and has a non-trivial expansion step that fills in the odd-indexed answers.

Algorithm: Scan

```
fun scan(f : (T,T) -> T, I : T, S : sequence<T>) -> (sequence<T>,T):
  match length(S) with:
  case 0: return [], I
  case 1: return [I], S[0]
  case _:
    B = parallel [f(S[2*i], S[2*i+1]) for i in 0...|S|/2-1]
      + ([S[|S|-1]] if |S|%2 == 1 else [])
    R, total = scan(f, I, B)
    return tabulate(fn i => R[i/2] if i%2==0
                      else f(R[i/2], S[i-1]), |S|), total
```

Theorem: Cost of Scan

Assuming f can be evaluated in $O(1)$ time, scan costs $O(|S|)$ work and $O(\log |S|)$ span.

Proof. Let $n = |S|$. The contraction step pairs up and sums adjacent elements, which costs $O(n)$ work and $O(1)$ span. The expansion step performs a tabulate of length n to produce the answer, which also costs $O(n)$ work and $O(1)$ span. The recursion step recursively calls scan on a sequence of half the length, so the work and span recurrence relations are:

$$W(n) = W\left(\frac{n}{2}\right) + O(n),$$

$$S(n) = S\left(\frac{n}{2}\right) + O(1).$$

These are the same recurrences as contraction-based reduce! Therefore, the contraction-based scan algorithm has the same cost bounds of $O(n)$ work and $O(\log n)$ span. $\square$

Note that just like with reduce, this assumes that f can be evaluated in constant time; if f has different cost bounds, then a separate analysis is necessary to compute the work and span for that particular call to the scan function. There is no formula that works in general.

3 Associative Functions

Recall from last lecture that reduce is only well-defined when the combining function f is *associative* and the base-case value I is an *identity value*.

Definition: Associative

A binary operator $f : (T, T) \rightarrow T$ is associative if for all values $x, y, z \in T$,

$$f(f(x, y), z) = f(x, f(y, z)).$$

Definition: Identity

A value $I \in T$ is an identity for a binary operator $f : (T, T) \rightarrow T$ if for every value x ,

$$f(I, x) = f(x, I) = x.$$

Since `scan` generalizes `reduce`, it also has these preconditions on f and I .

3.1 Examples of Associative Functions

There are many useful combining functions that are associative, including non-trivial ones that take longer than constant time because they operate not on individual numbers but on structured values such as sequences, sets, matrices, permutations, and functions!

- `min`, `max`, addition, multiplication, **and**, **or**, `xor`. These are all associative for integer types. Addition and multiplication are associative over the real numbers, but floating-point implementations are not exactly associative because of rounding. Nonetheless we will often accept rounding errors and use `reduce` or `scan` with floating-point numbers since it is still useful.
- `append` and `merge` for sequences. `append` is associative because concatenating many sequences results in the same final sequence regardless of the intermediate grouping used. `merge` is similar because merging several sorted sequences gives the same final sorted sequence regardless of how the merges are grouped.
- `union`, `intersection`, and `symmetric_difference` of sets! Grouping the operations differently does not change which elements belong to the resulting set.
- Matrix multiplication, composition of permutations, and function composition.

Remark: Associative versus Commutative

Associative and commutative **are not the same thing**, and neither implies the other. Commutative means that you can switch the *argument order*, i.e., $f(x, y) = f(y, x)$ for all x, y , while associative means you can change the grouping/parenthesization of multiple applications of f . Some operations are both, but you can not assume this in general. Can you think of which operations above are commutative and which are not?

3.2 An Interesting Example: `copy_right`

Consider the following binary function called `@` that combines optionals:

```
fun copy_right(a : optional<T>, b : optional<T>) -> optional<T>:
  match b with:
  case SOME(_): return b
  case _: return a
```

This function returns b if it is not a `NONE`, otherwise it returns a , so intuitively it is “copy the rightmost one if it’s not `NONE`,” hence its name. For example:

$f(\text{SOME}(2), \text{SOME}(5)) \rightarrow \text{SOME}(5)$	$f(\text{NONE}, \text{SOME}(5)) \rightarrow \text{SOME}(5)$
$f(\text{SOME}(2), \text{NONE}) \rightarrow \text{SOME}(2)$	$f(\text{NONE}, \text{NONE}) \rightarrow \text{NONE}$

Its neither immediately clear what this function is useful for, nor whether it has any interesting properties, but it will turn out to be very useful because despite not initially looking like it, this function is actually associative!

Theorem: Associativity of copy_right

The function `copy_right` is associative, with `NONE` as an identity value.

Proof. There are eight possible logical cases for x, y, z : each of them can be either `SOME(x)` or `NONE` (note that if they are `SOME(x)`, the value x is irrelevant to whether a or b is returned). The proof simply (tediously) enumerates all such eight cases. Let $f = \text{copy_right}$.

x	y	z	$f(x, y)$	$f(y, z)$	$f(f(x, y), z)$	$f(x, f(y, z))$
<code>SOME(x)</code>	<code>SOME(y)</code>	<code>SOME(z)</code>	<code>SOME(y)</code>	<code>SOME(z)</code>	<code>SOME(z)</code>	<code>SOME(z)</code>
<code>SOME(x)</code>	<code>SOME(y)</code>	<code>NONE</code>	<code>SOME(y)</code>	<code>SOME(y)</code>	<code>SOME(y)</code>	<code>SOME(y)</code>
<code>SOME(x)</code>	<code>NONE</code>	<code>SOME(z)</code>	<code>SOME(x)</code>	<code>SOME(z)</code>	<code>SOME(z)</code>	<code>SOME(z)</code>
<code>SOME(x)</code>	<code>NONE</code>	<code>NONE</code>	<code>SOME(x)</code>	<code>NONE</code>	<code>SOME(x)</code>	<code>SOME(x)</code>
<code>NONE</code>	<code>SOME(y)</code>	<code>SOME(z)</code>	<code>SOME(y)</code>	<code>SOME(z)</code>	<code>SOME(z)</code>	<code>SOME(z)</code>
<code>NONE</code>	<code>SOME(y)</code>	<code>NONE</code>	<code>SOME(y)</code>	<code>SOME(y)</code>	<code>SOME(y)</code>	<code>SOME(y)</code>
<code>NONE</code>	<code>NONE</code>	<code>SOME(z)</code>	<code>NONE</code>	<code>SOME(z)</code>	<code>SOME(z)</code>	<code>SOME(z)</code>
<code>NONE</code>	<code>NONE</code>	<code>NONE</code>	<code>NONE</code>	<code>NONE</code>	<code>NONE</code>	<code>NONE</code>

Furthermore, note that $f(\text{NONE}, x) = x$ and $f(x, \text{NONE}) = x$, so `NONE` is an identity value. □

4 Reduce with Custom Associative Functions

We have recently had a lot of practice with divide and conquer. Last lecture, we implemented `reduce` as a parallel divide-and-conquer algorithm like so:

```
fun reduce(f : (T, T) -> T, I : T, S : sequence<T>) -> T:
  match length(S) with:
  case 0: return I
  case 1: return S[0]
  case _:
    L, R = split_mid(S)
    Lres, Rres = parallel (reduce(f, I, L), reduce(f, I, R))
    return f(Lres, Rres)
```

This implementation takes a sequence of type T , an arbitrary associative combining function $f : (T, T) \rightarrow T$, and a corresponding identity value I . You have also seen several parallel divide-and-conquer algorithms in recitation and your first lab. You may have noticed that the code looks extremely similar; lets take a closer look.

4.1 SMCSS

Recall MCSSLab, where we implemented the “strengthened MCSS” problem using divide-and-conquer. The algorithm looked something like this (but in your favourite language):

```
fun smcss(S : sequence<int>) -> (int,int,int,int):
  match length(S) with:
  case 0: return (0,0,0,0)
  case 1:
    m = max(0, A[0])
    return (m, m, m, A[0])
  case _:
    L, R = split_mid(S)
    (m1,p1,s1,t1), (m2,p2,s2,t2) = parallel (smcss(L), smcss(R))
    return (max(s1 + p2, m1, m2),
            max(p1, t1 + p2),
            max(s2, t2 + s1),
            t1+t2)
```

The code is also divide-and-conquer, but beyond that it has striking similarities to our implementation of reduce. It has an empty sequence case, a length 1 base case, and a general case which splits the input into two halves, recursively solves the problem on those halves, and then combines those answers.

What is remarkable is that smcss is actually just a **special case of reduce**. If we factor out the combining-the-two-halves logic into its own function,

```
type sums = (int,int,int,int)

fun combine_smcss((m1,p1,s1,t1) : sums, (m2,p2,s2,t2) : sums) -> sums:
  return (max(s1 + p2, m1, m2),
          max(p1, t1 + p2),
          max(s2, t2 + s1),
          t1+t2)
```

Then we can observe that we can implement smcss as a reduce like so:

Algorithm: SMCSS as Reduce

```
fun smcss(S : sequence<int>) -> (int,int,int,int):
  fun base(x : int): return (max(0,x),max(0,x),max(0,x),x)
  return reduce(combine_smcss, (0,0,0,0), map(base, S))
```

That is, we use a **map** to solve the base cases by converting each individual input element into its corresponding base case value, then we perform the recursive divide-and-conquer by simply passing combine_smcss to reduce as the binary operation!

This is not an isolated coincidence. Let’s look at another example.

4.2 Parenthesis Matching

In the *Parenthesis Matching* recitation, we continued to practice divide-and-conquer by giving an efficient parallel implementation of the parenthesis matching problem. Specifically, this strengthened parenthesis matching problem counted the number of excess left and right parenthesis in a given subexpression.

```
fun excessParens (ps : sequence<Paren>) -> (int,int):
  match length(ps) with:
  case 0: return (0, 0)
  case 1:
    if (ps[0] == L): return (0, 1)
    else: return (1, 0)
  case _:
    a, b = split_mid(ps)
    (i,j), (k,l) = parallel (excessParens (a), excessParens (b))
    if (j <= k): return (i + k - j, l)
    else: return (i, l + j - k)
```

Again, we see the same pattern emerge: an empty case, a length-1 sequence base case, and a general case which splits the input into two and recursively solves the problem on the two halves before combining those results together into a final result.

We can once again factor out the “combination” logic into its own function.

```
fun combine_paren((i : int, j : int), (k : int, l : int)) -> (int, int):
  return (i + k - j, l) if j <= k else (i, l + j - k)
```

Using this combination function, we can once again rewrite this entire algorithm as a reduce!

Algorithm: excessParens as Reduce

```
fun excessParens(p : sequence<Paren>) -> (int,int):
  fun base(x : Paren): return (0,1) if x == L else (1,0)
  return reduce(combine_paren, (0,0), map(base, p))
```

4.3 Reduce as “Generic Divide-and-Conquer”

This observation gives us a recipe to convert many (but not all) divide-and-conquer algorithms into much shorter code by using reduce. In general, this technique is applicable whenever the “divide” step of the divide-and-conquer is trivial: it must simply split the input in half and recurse. If the algorithm needs additional steps before recursion, it isn’t applicable.

For example, Quicksort doesn’t just break the input into two halves and recurse, it partitions the input into those elements less than the pivot and those greater than the pivot, then recursively sorts those, so its not in the right format.

Similarly the algorithm for merge is not in the right format, because it has to use a binary search to locate the correct split point for the second sequence. Or, perhaps more obviously, its also not applicable because it operates on *two* sequences as input, and reduce only takes one.

We can see, however, that many algorithms are, like the `smcss` problem and the `excessParen` problem. Generally, if a divide-and-conquer algorithm is in the following form:

```
fun algo(S : sequence<T>):  
  match length(S) with:  
  case 0: return empty  
  case 1: return base(S[0])  
  case _:  
    L, R = split_mid(S)  
    Lres, Rres = parallel (algo(L), algo(R))  
    return combine(Lres, Rres)
```

where `combine` is associative and `empty` is an identity value for it, then it is equivalent to:

Algorithm: “Generic” Divide-and-Conquer

```
fun algo(s : sequence<T>):  
  return reduce(combine, empty, map(base, s))
```

Note that `map` being used to handle the base case makes this strictly more generic than a `reduce` on its own which would require that the output of the algorithm be the same as the input type `T` of the sequence. Using `map` allows us to make the output type different, which is powerful and necessary in many applications of this technique. You can think of the `map` as setting up the “strengthening” by converting each element of the input sequence to its strengthened form to then be combined by the combining function.

4.4 Sorting by Reducing with Merge

Recall the `merge` function from last lecture, which takes two sorted sequences and produces a sorted sequence containing the elements of both. What can we do with it? It turns out, more than you’d think. Although you might not realize it at first, **the merge function is associative!**

Claim: Merge is Associative

The merge operation given two sorted sequence is an associative operation.

Proof. Since `merge` must produce a sorted sequence given any sorted inputs, the result of merging three sequences x, y, z must always be sorted regardless of the order that the merges are applied. If any order produced a different output, it would not be sorted, and hence the output must be the same for any order. $\square$

Since `merge` is associative, given our newfound knowledge from earlier, our first instinct should be to put it in **reduce** and see what happens, because that would be interesting!

4.4.1 Parallel MergeSort as reduce merge

Since `merge` requires sequences as its input, we can treat the base case as being mapping a single element to a *singleton* sequence, i.e., a sequence containing that one element. A sequence of one element is considered sorted, so this is a valid input for `merge`.

Lastly, the identity value for a merge is the empty sequence, so we could therefore write the code `reduce(merge, [], map(singleton, s))` where `singleton = fn(s) => [s]`. This is valid because it meets all the requirements of reduce, but what does it actually do? It takes all the elements of a sequence, puts them into their own singleton sequences, and repeatedly merges them until they form a single sorted sequence.

Hang on, merging the elements of the input together until they are one sorted sequence? That's literally just **Merge Sort**! Through the magic of reduce, we have managed to write Merge Sort in just one line of code. That's pretty cool.

Theorem: Cost of Parallel MergeSort

Parallel MergeSort, which can be implemented by the following line of code:

```
fn(s : sequence<T>) => reduce(merge, [], map(singleton, s))
```

runs in $O(n \log n)$ work and $O(\log^2 n)$ span, assuming we use the $O(\log n)$ span merge.

Proof. MergeSort recurs on sequences of size $n/2$ and performs $O(n)$ work at the root for the merge operation. The work recurrence is therefore

$$W(n) = 2W\left(\frac{n}{2}\right) + O(n).$$

By the brick method, this is a balanced recurrence since the work at the parent is n , and the two children each perform $n/2$ work which sums to n . The recurrence has $O(\log n)$ levels which perform $O(n)$ work per level, and hence the work is $O(n \log n)$.

Correspondingly the span recurrence (assuming merge costs $O(\log n)$ span) is

$$S(n) = S\left(\frac{n}{2}\right) + O(\log n).$$

By the brick method, this is also a balanced recurrence since the cost of the parent is $\log n$, and the cost of the child is $\log n - 1$, which is not a constant factor smaller or larger. The recurrence has $O(\log n)$ levels and hence the span is $S(n) = O(\log^2 n)$. $\square$

4.4.2 InsertionSort as fold merge

Finally, what if we do a **left fold** instead of a reduce? We know that it must produce the same answer since a fold and a reduce always compute the same result if the function is associative, it just might compute it in a different way.

```
fn(s : sequence<T>) => fold_left(merge, [], map(singleton, s))
```

This function takes a sequence and one-by-one, merges one additional new element into a sorted prefix. Hang on, that's just called **Insertion Sort**! With the magic of higher-order functions, we can write both Insertion Sort and MergeSort in a single line of code (well, we also have to write parallel merge which is many more than one line of code, but still.) Pretty cool!

5 Scan With Custom Associative Functions

Just as reduce can be used with non-trivial combining functions, so too can scan. Scan is even more powerful because it computes strictly more information than reduce. Here is one interesting problem we can solve. What makes it really interesting is that it sounds like a *very sequential* problem with no obvious parallel solution, but with scan, it becomes simple.

Problem: Previous SOME

Given a sequence S of `optional<T>`, for every position $0 \leq i < |S|$. compute the right-most SOME (i.e., non-NONE) value that occurs before position i .

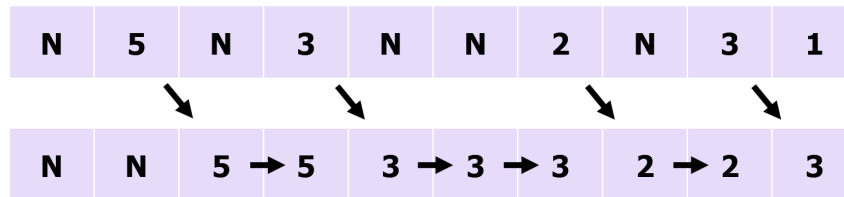
For example, given the sequence

[NONE, SOME(5), NONE, SOME(3), NONE, NONE, SOME(2), NONE, SOME(3), SOME(1)],

the desired output is

[NONE, NONE, SOME(5), SOME(5), SOME(3), SOME(3), SOME(3), SOME(2), SOME(2), SOME(3)].

Notice that what happens is that the SOMEs “propagate” to the right and overwrite the Nones.



A simple sequential solution would be to loop left-to-right and just take the previous value if the current value is NONE, otherwise take the current value. This is, coincidentally, exactly a left fold over the binary operation “copy_right” that we saw earlier:

```
fun copy_right(a : optional<T>, b : optional<T>) -> optional<T>:
  match b with:
  case SOME(_): return b
  case _: return a
```

It's easy to see why this solution works sequentially, but it sounds like it obviously isn't going to work in parallel because it has to loop left to right. But, the moment we realize/remember that `copy_right` is associative, we know that we can use it with scan and hence solve it in parallel!

Algorithm: Previous SOME

```
fun previous_some(S : sequence<optional<T>>) -> sequence<optional<T>>:
  propagated, _ = scan(copy_right, NONE, S)
  return propagated
```

This gives us an $O(n)$ work and $O(\log n)$ span solution to find the previous non-NONE option at every position in the sequence. This may seem contrived, but this technique of propagating information left-to-right down a sequence has useful applications in several parallel algorithms. You'll want to remember it, because you will see one in your very next lab!