Algorithms for Sequences

1 Arrays and Imperative Parallelism

Recall from Lecture One, the sequence abstract data type (ADT).

Interface: The Sequence ADT

A sequence represents a finite, ordered collection of elements that supports efficient (i.e., constant-time) random access. Conceptually, a sequence behaves like an array or a contiguous view into an array.

Sequence Access:

- `nth : (S : sequence<T>, i : int) -> T`
  Return the i^{th} element (zero-indexed) of the sequence S.
- `length : (S : sequence<T>) -> int`
  Return the length (number of elements in) the sequence S.

The `nth` function is usually abbreviated as `S[i]`, and `length` is abbreviated as `|S|`.

Subsequence Views:

- `subseq : (S : sequence<T>, i : int, k : int) -> sequence<T>`
  Returns a *view* of the subsequence of S starting at index i with length k.

The function `subseq` does not copy elements, otherwise it would be inefficient.

Any concrete type that implements this interface is considered a sequence. Since a `sequence` is, at a high level, a thing that behaves like an array, the most obvious way to implement the sequence ADT is using arrays!

So far in class we have just been assuming that functions like `tabulate` and our other sequence functions return array-sequences.

- `tabulate : ((int -> T), int) -> ArraySequence<T>`

In this lecture, we will go a bit more under the hood and talk about how `ArraySequence` and many important sequence algorithms are implemented.

1.1 Arrays

Arrays work slightly differently in different programming languages, but for the most part they support the following set of fundamental operations with some syntax in some form or another:

- `allocate<T>(n)`: allocate an array of length n of type `T`
- `A[i]`: return the i^{th} element of A
- `|A|`: return the length of A
- `A[i] ← x`: write the value `x` to the i^{th} element of A

All of these operations runs in $O(1)$ time. In most languages, `|A|` would be written as `A.length` or `A.size()` or something similar, but we'll keep this shorthand to make our pseudocode nicer.

And of course, to make everything work under the hood, we are going to need to assume that arrays are *mutable*. The fourth operation in our list of primitive array operations allows us to write a value to an element of the array.

1.2 Imperative Parallel Loops

Thus far, we have written and described parallelism in algorithms using only functional constructs; either parallel list comprehension (our pseudocode for `tabulate`), which evaluates a (pure) function over many indices in parallel and returns the results as a sequence, or parallel tuple evaluation, which evaluates several expressions in parallel and returns their results.

When we use these constructs to express parallelism, by default we try to avoid side-effects since this makes the code harder to reason about and more error prone. To implement the low level parallel operations that our algorithms rely on (for example, to implement tabulate itself), we will however need to write *imperative* parallel code, i.e., parallel code with side effects.

The standard abstraction used and supported by most imperative parallel programming libraries is the **parallel for loop**. A parallel for loop evaluates the iterations of its loop body in parallel. Since it returns no value, its purpose is to execute a loop that produces side effects.

Definition: Parallel For Loop

A parallel for loop executes its body for every value of its range in parallel. In our pseudocode, we denote a parallel for loop, where `e(i)` is some (impure) function of i, as:

```
parallel for i in ...:
  e(i)
```

Since the iterations of the parallel for loop execute in parallel, its work and span are

$$W = \sum_i W(e(i)), \qquad S = \max_i S(e(i)).$$

Now that we have a parallel construct that can execute imperative code, we are vulnerable to a new kind of bug that we must be extremely careful to avoid, a **data race**.

Definition: Data Race

A *data race* occurs when there are two unsynchronized parallel operations on the same memory location, and at least one of the operations is a write.

Note that two parallel *reads* of the same memory location is not a data race. Any access that involves a write must be treated with care.

The precise meaning of "synchronization" depends on the programming language, its memory model, and sometimes even the hardware. These details are beyond the scope of this class. As a result, we will adopt a deliberately conservative rule: *All parallel operations in this class are treated as unsynchronized.*

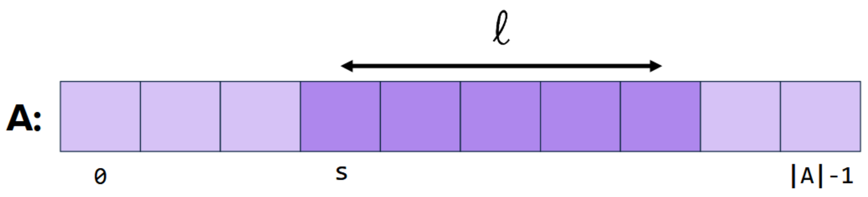
Under this rule, **any** pair of parallel operations that access the same memory location, where at least one is a write, constitutes a data race. This cannot happen in a purely functional program, since values are never mutated. However, it *can* happen in imperative parallel constructs such as parallel **for** loops.

In this class, the presence of a data race means the algorithm is *incorrect*. Programs with data races are considered to have undefined behavior and are not valid algorithms.

2 A Data Type for ArraySequence

Because we want our implementation of subseq (extract a contiguous subsequence) to be fast, our data structure for ArraySequences will keep the actual array, along with two additional integers. These are the starting point of the subsequence, and the length of it. So we have:

```
type ArraySequence<T> =
  {
    A: Array<T>
    s: int
    l: int
  }
```



s is the starting index of the subsequence in the array A, and ℓ is the length of the subsequence. When we create an ArraySequence initially, s will be zero and l will be the length of the array.

2.1 nth and length

`nth` and `length` are straightforward to implement given the `ArraySequence` data type.

Algorithm: nth

```
fun nth (S : ArraySequence<T>, i : int) -> T:
  (A,s,_) = S
  return A[s+i]
```

Algorithm: length

```
fun length(S : ArraySequence<T>) -> int:
  (_, _, l) = S
  return l
```

Both of these run in constant time (work and span).

2.2 subseq

Since our `ArraySequence` data type stores an offset and length, implementing `subseq` trivial; we just add to the start and set the new length while keeping the same underlying array.

Algorithm: Subseq

```
fun subseq (S: ArraySequence<T>, s: int, l: int) -> ArraySequence<T>:
  (A,s',l') = S
  return (A,s+s',l)
```

`subseq` also runs in constant time (work and span).

2.3 tabulate

Since `tabulate` is our key primitive for *creating* a new `ArraySequence`, it actually has to allocate the memory for the array and fill it with its values. We can do this using our assumed `allocate` primitive and a parallel for loop to populate the values.

Algorithm: Tabulate

```
fun tabulate (f : (int -> T), n : int) -> ArraySequence<T>:
  R = allocate<T>(n)
  parallel for i in 0...n-1:
    R[i] ← f(i)
  return (R, 0, n)
```

Is there a data race? No! Because each of the parallel executions of the loop body operates on a different element of R. (We assume that the function f is pure, i.e., has no side effects.)

The work and span of `tabulate` are

$$W = \sum_{0 \le i < n} W(f(i)), \qquad S = \max_{0 \le i < n} S(f(i)).$$

3 Sequence Functions

With `ArraySequence` as a concrete implementation of the `sequence` ADT, we can now implement a range of useful algorithms on sequences. We have already seen how to implement a few of these in past lectures (most notably `reduce`, `scan`, and `merge`, and also `sort` by doing a reduce over the `merge` function as an implementation of MergeSort!)

- `map(f,S)`: applies the function f to every element of S, returning a sequence of results
- `append(S1, S1)`: concatenate the two given sequences into a single sequence
- `filter(p, S)`: return a sequence containing the elements of S that satisfy the predicate f
- `flatten(S)`: given a nested sequence, return a sequence of the values of the inner sequences
- `reduce(f, I, S)`: compute the sum of S with respect to the function f and identity I
- `scan(f, I, S)`: compute the prefix sums of S with respect to the function f and identity I
- `merge(S1, S2)`: given two sorted sequences, return a sorted sequence of their elements
- `sort(S)`: return a copy of the given sequence with the elements in sorted order

Some of these are straightforward applications of `tabulate`, while others will require more complicated algorithms, possibly involving some of the other functions.

3.1 Map

The `map` function applies a given (pure) function to the elements of a given sequence and returns a new sequence consisting of the results. This can be implemented by just calling `tabulate` using a function that maps from an index `i` to `f(S[i])`.

Algorithm: Map

```
fun map (f : (T -> U),  S : sequence<T>) -> ArraySequence<U>:
  return tabulate(fn i => f(S[i]), |S|)
```

The work and span of `map` are

$$W = \sum_{x \in S} W(f(x)), \qquad S = \max_{x \in S} S(f(x)).$$

3.2 Append

The `append` function takes two sequences and concatenates them together. This can also be implemented as an application of `tabulate`. We just case on whether the index `i` is within the first or the second sequence.

Algorithm: Append

```
fun append (A : sequence<T>, B : sequence<T>) -> ArraySequence<T>:
  return tabulate (fn i => (A[i] if i<|A| else B[i-|A|]), |A|+|B|)
```

The work and span of `append` are

$$W = O(|A| + |B|), \qquad S = O(1).$$

3.3 Filter

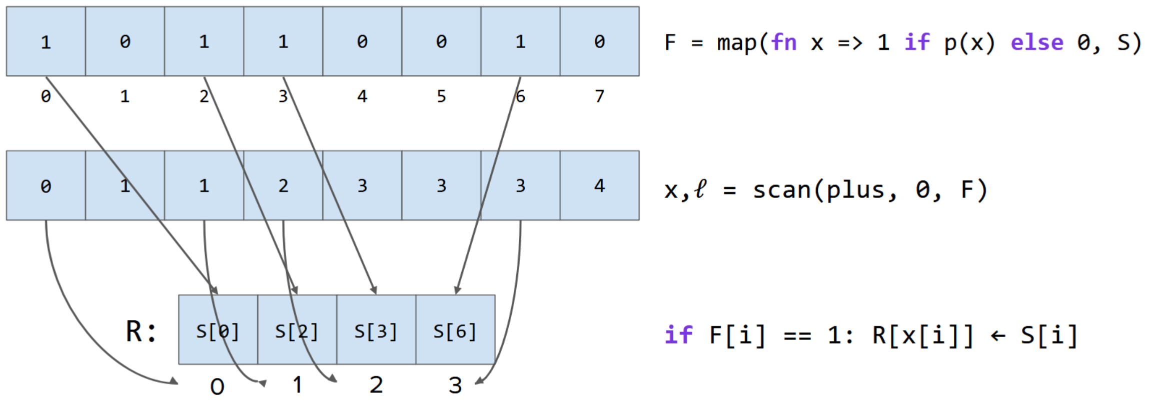
`filter(p, S)` returns a sequence consisting of the elements of S which satisfy the predicate p. The relative order of the elements is preserved. The type signature of filter is:

```
filter : (p : (T -> bool), S : sequence<T>) -> ArraySequence<T>
```

For example `filter (fn x => x < 5, [7,1,3,11,7,2])` returns `[1,3,2]`

Our goal is to do this in linear work and logarithmic span (that's the interesting part). Suppose we have a sequence of eight elements as input to `filter`, and the ones that satisfy the predicate are the $0^{\text{th}}, 2^{\text{nd}}, 3^{\text{rd}}, 6^{\text{th}}$. The trick is to figure out *where* in the output these elements occur.

This is done by the following key observation: the position of a surviving element in the output is exactly equal to the *number of surviving elements that occur before it*. Counting the number of elements *before* each elements sounds like a prefix sum, because it is, so we can use `scan`!



The idea is to first map each element x to 1 if it survives (i.e., if $p(x)$ returns **True**) or 0 otherwise. These indicator variables are then scanned to produce the prefix sums, i.e., exactly how many surviving elements occur before each element. These prefix sums then give the algorithm the locations to which to write the survivors in the output array, which can be done in parallel since no two elements will ever occupy the same position, so no data race will occur.

Algorithm: Filter

```
fun filter (p : (T -> bool), S : Sequence<T>) -> ArraySequence<T>:
  F = map (fn x => 1 if p(x) else 0, S)
  x,l = scan(plus, 0, F)
  R = allocate<T>(l)
  parallel for i in 0...|S|-1:
    if F[i] == 1:
      R[x[i]] ← S[i]
  return (R,0,l)
```

The work and span of `filter` are given by the sums of the work and span of the `map` and `scan` calls performed, which depend on the work and span of the predicate p:

$$W = \sum_{x \in S} W(p(x)), \qquad S = O(\log |S|) + \max_{x \in S} S(p(x)).$$

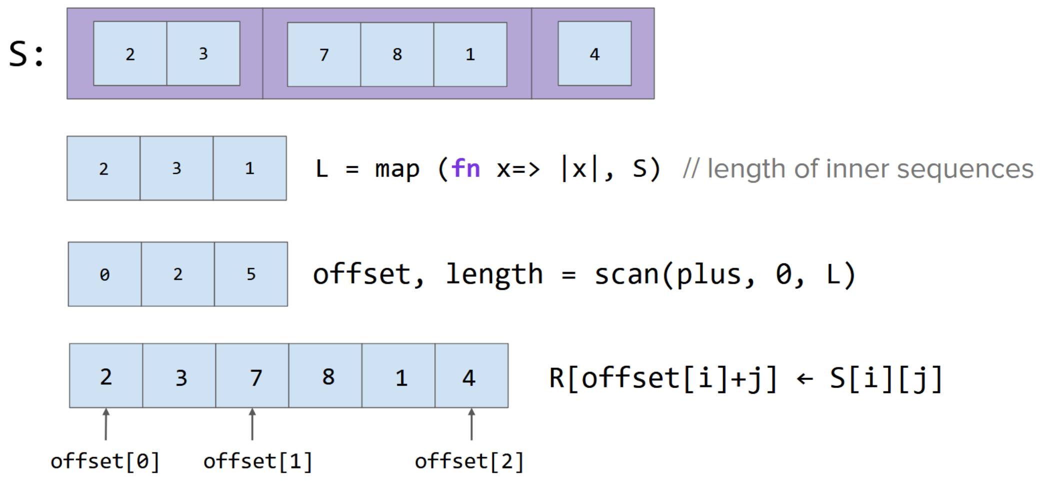
3.4 Flatten

The `flatten` function takes a sequence of sequences (i.e., a 2D nested sequence) and produces a new sequence consisting of all the elements of the inner sequences in the same relative order. In other words, it concatenates all of the inner sequences. The type of flatten is:

```
flatten: (S : sequence<sequence<T>>) -> ArraySequence<T>
```

For example, `flatten([[2,3],[7,8,1],[4]])` returns `[2,3,7,8,1,4]`.

The fundamental problem here is similar to that of `filter`: for each element we need to figure out its position in the output array. The solution is therefore also very similar to that of `filter`. The position of an element in the output array is its position in its inner (input) array *plus* the lengths of all the inner sequences that occur before it. This is yet *another* scan!


S:
2
3
7
8
1
4
2
3
1
L = map (fn x=> |x|, S) // length of inner sequences
0
2
5
offset, length = scan(plus, 0, L)
2
3
7
8
1
4
R[offset[i]+j] ← S[i][j]
offset[0]
offset[1]
offset[2]

The idea is to use `scan` to compute the prefix sums of the lengths of the inner sequences. These tell you where in the output the elements of each sequence should go. Then, the algorithm can simply copy each element of the input sequences to its destination in parallel. Similar to `filter`, no two input elements share the same output position so no data race occurs.

Algorithm: Flatten

```
fun flatten (S : sequence<sequence<T>>) -> ArraySequence<T>:
  L = map (fn x => |x|, S)
  offset, length = scan(plus, 0, L)
  R = allocate<T>(length)
  parallel for i in 0...|S|-1:
    parallel for j in 0...L[i]-1:
      R[offset[i]+j] ← S[i][j]
  return (R, 0, length)
```

The work and span of `flatten` are:

$$W = O\left(\sum_{x\in S}(1+|x|)\right), \qquad S = O(\log|S|).$$