Chapter 18

The Sequence Abstract Data Type

Sequences are one of the most prevalent ADTs (Abstract Data Types) used in this book, and more generally in computing. In this chapter, we present the interface of an ADT for sequences, describe the semantics of the functions in the ADT, and define the notation we use in this book for sequences.

1 The Abstract Data Type

Data Type 18.1 (Sequences)**.** Define booleans as

$$\mathbb{B} = \{\texttt{true}, \texttt{false}\},$$

and orders as

$$\mathcal{O} = \{less, greater, equal\}.$$

For any element type α, the α- ***sequence data type*** is the type $\mathbb{S}_\alpha$ consisting of the set of all α sequences, and the following values and functions on $\mathbb{S}_\alpha$.

length	:	$\mathbb{S}_\alpha \to \mathbb{N}$
nth	:	$\mathbb{S}_\alpha \to \mathbb{N} \to \alpha$
empty	:	$\mathbb{S}_\alpha$
singleton	:	$\alpha \to \mathbb{S}_\alpha$
tabulate	:	$(\mathbb{N} \to \alpha) \to \mathbb{N} \to \mathbb{S}_\alpha$
map	:	$(\alpha \to \beta) \to \mathbb{S}_\alpha \to \mathbb{S}_\beta$
subseq	:	$\mathbb{S}_\alpha \to \mathbb{N} \to \mathbb{N} \to \mathbb{S}_\alpha$
append	:	$\mathbb{S}_\alpha \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha$
filter	:	$(\alpha \to \mathbb{B}) \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha$
flatten	:	$\mathbb{S}_{\mathbb{S}_\alpha} \to \mathbb{S}_\alpha$
update	:	$\mathbb{S}_\alpha \to (\mathbb{N} \times \alpha) \to \mathbb{S}_\alpha$
inject	:	$\mathbb{S}_\alpha \to \mathbb{S}_{\mathbb{N}\times\alpha} \to \mathbb{S}_\alpha$
isEmpty	:	$\mathbb{S}_\alpha \to \mathbb{B}$
isSingleton	:	$\mathbb{S}_\alpha \to \mathbb{B}$
collect	:	$(\alpha \times \alpha \to \mathcal{O}) \to \mathbb{S}_{\alpha\times\beta} \to \mathbb{S}_{\alpha\times\mathbb{S}_\beta}$
iterate	:	$(\alpha \times \beta \to \alpha) \to \alpha \to \mathbb{S}_\beta \to \alpha$
reduce	:	$(\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to \alpha$
scan	:	$(\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to (\mathbb{S}_\alpha \times \alpha)$

where the semantics of the values and functions are described in this chapter.

Syntax 18.2 (Sequence Comprehensions)**.** Inspired by mathematical notation for sequences, we use a "sequence comprehensions" notation as defined below. In the definition,

- i is a variable ranging over natural numbers,
- x is a variable ranging over the elements of a sequence,
- e is a SPARC expression,
- e_n and e'_n are SPARC expressions whose values are natural numbers,
- e_s and e'_s are SPARC expressions whose values are a sequence,
- p is a SPARC pattern that binds one or more variables.

$\lvert e_s \rvert$	$\equiv$	*length* e_s
$e_s[i]$	$\equiv$	*nth* e_s
$\langle\ \rangle$	$\equiv$	*empty*
$\langle\, e\, \rangle$	$\equiv$	*singleton* e
$\langle\, e : 0 \le i < e_n \,\rangle$	$\equiv$	*tabulate* $(\texttt{lambda}\; i \,.\, e)\; e_n$
$\langle\, e : p \in e_s \,\rangle$	$\equiv$	*map* $(\texttt{lambda}\; p \,.\, e)\; e_s$
$\langle\, p \in e_s \mid e \,\rangle$	$\equiv$	*filter* $(\texttt{lambda}\; p \,.\, e)\; e_s$
$e_s[e_n, \cdots, e_{n'}]$	$\equiv$	*subseq* $(e_s, e_n, e'_n - e_n + 1)$
$e_s \mathrel{++} e'_s$	$\equiv$	*append* $e_s\; e'_s$

2 Basic Functions

Definition 18.3 (Length and indexing)**.** Given a sequence a, *length* a, also written $|a|$, returns the length of a (i.e., number of elements). The function *nth* returns the element of a sequence at a specified index, e.g. *nth* a 2, written $a[2]$, returns the element of a with rank 2. If the element demanded is out of range, the behavior is undefined and leads to an error.

Definition 18.4 (Empty and singleton)**.** The value *empty* is the empty sequence, $\langle\ \rangle$. The function *singleton* takes an element and returns a sequence containing that element, e.g., *singleton* 1 evaluates to $\langle\, 1\, \rangle$.

Definition 18.5 (Functions *isEmpty* and *isSingleton*)**.** To identify trivial sequences such as empty sequences and singleton sequences, which contain only one element, the interface provides the functions *isEmpty* and *isSingleton*. The function *isEmpty* returns `true` if the sequence is empty and `false` otherwise. The function *isSingleton* returns `true` if the sequence consists of a one element and `false` otherwise.

3 Tabulate

Definition 18.6 (Tabulate)**.** The function *tabulate* takes a function f and an natural number n and produces a sequence of length n by applying f at each position. The function f can be applied to each element in parallel. We specify *tabulate* as follows

$$\begin{array}{l} \mathit{tabulate}\ (f : \mathbb{N} \to \alpha)\ (n : \mathbb{N}) : \mathbb{S}_\alpha \\ \quad = \langle\, f(0), f(1), \ldots, f(n-1)\, \rangle\,. \end{array}$$

Syntax 18.7 (Tabulate)**.** We use the following syntax for *tabulate* function

$$\langle\, e : 0 \le i < e_n\, \rangle \equiv \mathit{tabulate}\ (\texttt{lambda}\ i \,.\, e)\ e_n,$$

where e and e_n are expressions, the second evaluating to an integer, and i is a variable. More generally, we can also start at any other index, as in:

$$\langle\, e : e_j \le i < e_n\, \rangle\,.$$

Example 18.1 (Fibonacci Numbers)**.** Given the function *fib* i, which returns the i^{th} Fibonacci number, the expression:

$$a = \langle\, \mathit{fib}\ i : 0 \le i < 9\, \rangle$$

is equivalent to

$$a = \mathit{tabulate}\ \ \mathit{fib}\ \ 9.$$

When evaluated, it returns the sequence

$$a = \langle\, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34\, \rangle\,.$$

4 Map and Filter

Mapping over a sequence or filtering out elements of a sequence that does not meet a desired condition are common tasks. The sequence ADT includes the functions map and *filter* for these purposes.

Definition 18.8 (Map)**.** The function map takes a function f and a sequence a and applies the function f to each element of a returning a sequence of equal length with the results. As with $\mathit{tabulate}$, in map, the function f can be applied to all the elements of the sequence in parallel.

We specify the behavior of map as follows

$$\begin{aligned} &\mathit{map}\ (f : \alpha \to \beta)\ (a : \mathbb{S}_\alpha) : \mathbb{S}_\beta \\ &\qquad = \{(i, f(x)) : (i, x) \in a\} \end{aligned}$$

or equivalently as

$$\mathit{map}\ (f : \alpha \to \beta)\ \langle\, a_1, \ldots, a_{n-1} \,\rangle : \mathbb{S}_\alpha) : \mathbb{S}_\beta = \langle\, f(a_1), \ldots, f(a_{n-1}) \,\rangle\,.$$

Syntax 18.9 (Map)**.** We use the following syntax for the map function

$$\langle\, e : p \in e_s \,\rangle \equiv \mathit{map}\ (\texttt{lambda}\ p\,.\,e)\ e_s,$$

where e and e_s are expressions, the second evaluating to a sequence, and p is a a pattern of variables (e.g., x or (x, y)).

Definition 18.10 (Filter)**.** The function *filter* takes a Boolean function f and a sequence a as arguments and applies f to each element of a. It then returns the sequence consisting exactly of those elements of $s \in a$ for which $f(s)$ returns true, while preserving the relative order of the elements returned.

We specify the behavior of *filter* as follows

$$\begin{aligned} &\mathit{filter}\ (f : \alpha \to \mathbb{B})\ (a : \mathbb{S}_\alpha) : \mathbb{S}_\alpha = \\ &\qquad \{(|\{(j, y) \in a \mid j < i \wedge f(y)\}|, x) : (i, x) \in a \mid f(x)\}\,. \end{aligned}$$

As with map and $\mathit{tabulate}$, the function f in *filter* can be applied to the elements in parallel.

Syntax 18.11 (Filter Syntax)**.** We use the following syntax for the *filter* function

$$\langle\, x \in e_s \mid e \,\rangle \equiv \mathit{filter}\ (\texttt{lambda}\ x\,.\,e)\ e_s,$$

where e and e_s are expressions. In the syntax, note the distinction between the colon (:) and the bar (|). We use the colon to draw elements from a sequence for mapping and we use the bar to select the elements that we wish to filter.

We can map and *filter* at the same time:

$$\begin{aligned} \langle\, e : x \in e_s \mid e_f \,\rangle \quad \equiv \quad & \mathit{map}\ (\texttt{lambda}\ x\,.\,e) \\ & \qquad (\mathit{filter}\ (\texttt{lambda}\ x\,.\,e_f)\ e_s). \end{aligned}$$

What appears before the colon (if any) is an expression to apply each element of the sequence to generate the result; what appears after the bar (if there is any) is an expression to apply to each element to decide whether to keep it.

Example 18.2. The expression

$$\left\langle\, x^2 : x \in a \,\right\rangle$$

is equivalent to

$$\mathit{map}\ (\texttt{lambda}\ x\,.\,x^2)\ a.$$

Assuming $a = \langle\, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \,\rangle$ (from above), it evaluates to the sequence:

$$\langle\, 0, 1, 1, 4, 25, 64, 169, 441, 1156 \,\rangle\,.$$

Given the function $\mathit{isPrime}\ x$ which checks if x is prime, the expression

$$\langle\, x : x \in a \;\mid\; \mathit{isPrime}\ x \,\rangle$$

is equivalent to

$$\mathit{filter}\ \mathit{isPrime}\ a.$$

When evaluated, it returns the sequence $\langle\, 2, 5, 13 \,\rangle$.

5 Subsequences

Definition 18.12 (Subsequences)**.** The $\mathit{subseq}(a, i, j)$ function extracts a contiguous subsequence of a starting at location i and with length j. If the subsequence is out of bounds of a, only the part within a is returned. We can specify subseq as follows

$$\begin{aligned}&\mathit{subseq}\ (a : \mathbb{S}_\alpha)\ (i : \mathbb{N})\ (j : \mathbb{N}) : \mathbb{S}_\alpha \\ &\qquad = \{(k - i, x) : (k, x) \in a \mid i \le k < i + j\}\,.\end{aligned}$$

We use the following syntax for denoting subsequences

$$a[e_i \cdots e_j] \equiv \mathit{subseq}\ (a, e_i, e_j - e_i + 1).$$

Splitting sequences. As we shall see in the rest of this book, many algorithms operate inductively on a sequence by splitting the sequence into parts, consisting for example, of the first element and the rest, a.k.a., the ***head*** and the ***tail***, or the first half or the second half. We could define additional functions such as *splitHead*, *splitMid*, *take*, and *drop* for these purposes. Since all of these are easily expressible in terms of subsequences, we omit their discussion.

6 Append and Flatten

For constructing large sequences from smaller ones, the sequence ADT provides the functions *append* and *flatten*.

Definition 18.13 (Append)**.** The function $\mathit{append}\ (a, b)$ appends the sequence b after the sequence a. More precisely, we can specify *append* as follows

$$\begin{aligned}&\mathit{append}\ (a : \mathbb{S}_\alpha)\ (b : \mathbb{S}_\alpha) : \mathbb{S}_\alpha \\ &\qquad = a \cup \{(i + |a|, x) : (i, x) \in b\}\end{aligned}$$

We write $a \mathbin{++} b$ as a short form for $\mathit{append}\ a\ b$.

Example 18.3 (Append)**.** The *append* function

$$\langle\, 1, 2, 3 \,\rangle \mathbin{++} \langle\, 4, 5 \,\rangle$$

yields

$$\langle\, 1, 2, 3, 4, 5 \,\rangle\,.$$

Definition 18.14 (Flatten)**.** To append more than two sequences the $\mathit{flatten}\ a$ function takes a sequence of sequences and flattens them. For the input is a sequence $a = \langle\, a_1, a_2, \ldots, a_n \,\rangle$, *flatten* returns a sequence whole elements consists of those of all the a_i in order. We can specify *flatten* more precisely as follows

$$\begin{aligned}&\mathit{flatten}\ (a : \mathbb{S}_{\mathbb{S}_\alpha}) : \mathbb{S}_\alpha \\ &\qquad = \left\{ \left(i + \sum_{(k,c) \in a, k < j} |c|, x \right) \ :\ (i, x) \in b, (j, b) \in a \right\}.\end{aligned}$$

Example 18.4 (Flatten)**.** The *flatten* function

$$\mathit{flatten}\ \langle\, \langle\, 1, 2, 3 \,\rangle, \langle\, 4 \,\rangle, \langle\, 5, 6 \,\rangle \,\rangle$$

yields

$$\langle\, 1, 2, 3, 4, 5, 6 \,\rangle\,.$$

7 Update and Inject

Definition 18.15 (Update)**.** The function $\mathit{update}\ (a, (i, x))$, updates location i of sequence a to contain the value x. If the location is out of range for the sequence, the function returns the input sequence unchanged.

We specify *update* as follows

$$\begin{aligned}&\mathit{update}\ (a : \mathbb{S}_\alpha)\ (i : \mathbb{N}, x : \alpha) : \mathbb{S}_\alpha \\ &\qquad = \begin{cases} \{(j, y) : (j, y) \in a \mid j \neq i\} \cup \{(i, x)\} & \text{if } 0 \leq i < |a| \\ a & \text{otherwise.}\end{cases}\end{aligned}$$

Definition 18.16 (Inject)**.** To update multiple positions at once, we can use $inject$. The function $inject\ \ a\ b$ takes a sequence b of position-value pairs and updates each position with its associated value. If a position is out of range, then the corresponding update is ignored. If multiple positions are the same, the first update in the ordering of b take effect. We define the ***degree*** of the update sequence b as the maximum number of updates that target any position.

Example 18.5 (Update and Inject)**.** Given the string sequence

$$a = \langle \texttt{'the'}, \texttt{'cat'}, \texttt{'in'}, \texttt{'the'}, \texttt{'hat'} \rangle,$$

$$update\ a\ (1, \texttt{'rabbit'})$$

magically yields

$$\langle \texttt{'the'}, \texttt{'rabbit'}, \texttt{'in'}, \texttt{'the'}, \texttt{'hat'} \rangle$$

since position 1 is updated with $\texttt{'rabbit'}$. The expression

$$inject\ a\ \langle (4, \texttt{'log'}), (1, \texttt{'dog'}), (6, \texttt{'hog'}), (4, \texttt{'bog'}), (0, \texttt{'a'}) \rangle$$

yields

$$\langle \texttt{'a'}, \texttt{'dog'}, \texttt{'in'}, \texttt{'the'}, \texttt{'log'} \rangle$$

because position 0 is updated with $\texttt{'a'}$, position 1 with $\texttt{'dog'}$, and position 4 with $\texttt{'log'}$ (the first of the two updates is applied). Because two updates target position 4 and at most 1 update targets all the other positions, the degree of the update sequence is 2.

Definition 18.17 (Nondeterministic Inject)**.** To update multiple positions at once, we can also use nondetermistic inject $ninject$. The function $ninject\ \ a\ b$ takes a sequence b of position-value pairs and updates each position with its associated value. If a position is out of range, then the corresponding update is ignored. If multiple positions are the same, any one of the updates may take effect. The function $ninject$ may thus treat duplicate updates non-deterministically. Because nondeterministic inject does not insist on determinism of updates, it may be implemented more efficiently and in lower span.

Example 18.6 (Nondeterministic Inject)**.** Given the string sequence

$$a = \langle \texttt{'the'}, \texttt{'cat'}, \texttt{'in'}, \texttt{'the'}, \texttt{'hat'} \rangle,$$

the expression

$$ninject\ a\ \langle (4, \texttt{'log'}), (1, \texttt{'dog'}), (6, \texttt{'hog'}), (4, \texttt{'bog'}), (0, \texttt{'a'}) \rangle$$

could yield

$$\langle \texttt{'a'}, \texttt{'dog'}, \texttt{'in'}, \texttt{'the'}, \texttt{'log'} \rangle$$

since position 0 is updated with $\texttt{'a'}$, position 1 with $\texttt{'dog'}$, and position 4 with $\texttt{'log'}$ (the first of the two updates is applied). It could also yield

$$\langle \texttt{'a'}, \texttt{'dog'}, \texttt{'in'}, \texttt{'the'}, \texttt{'log'} \rangle$$

The entry with position 6 is ignored since it is out of range for a.

8 Collect

Definition 18.18 (Collect)**.** Given a sequence of ***key-value*** pairs, the operation *collect* "collects" together all the values for a given key. This operation is quite common in data processing, and in relational database languages such as SQL it is referred to as "Group by". The signature of *collect* is

$$\mathit{collect} : (\mathit{cmp} : \alpha \times \alpha \to \mathcal{O}) \to (a : \mathbb{S}_{\alpha\times\beta}) \to \mathbb{S}_{\alpha\times\mathbb{S}_\beta}.$$

Here the "order set" $\mathcal{O} = \{\mathit{less}, \mathit{equal}, \mathit{greater}\}$.

The first argument *cmp* is a function for comparing keys of type α, and must define a total order over the keys. The second argument a is a sequence of key-value pairs. The *collect* function collects all values in a that share the same key together into a sequence, ordering the values in the same order as their appearance in the original sequence.

Example 18.7 (Collect)**.** The following sequence consists of key-value pairs each of which represents a student and the classes that they take.

$$\begin{aligned}
\mathit{kv} = \langle &(\texttt{'jack'},\texttt{'15210'}), (\texttt{'jack'},\texttt{'15213'}) \\
&(\texttt{'mary'},\texttt{'15210'}), (\texttt{'mary'},\texttt{'15213'}), (\texttt{'mary'},\texttt{'15251'}), \\
&(\texttt{'peter'},\texttt{'15150'}), (\texttt{'peter'},\texttt{'15251'}), \\
&\ldots \\
\rangle&.
\end{aligned}$$

We can determine the classes taken by each student by using *collect cmp*, where *cmp* is a comparison function for strings

$$\begin{aligned}
\mathit{collect}\ \mathit{cmp}\ \mathit{kv} = \langle\ &(\texttt{'jack'}, \langle\texttt{'15210'},\texttt{'15213'},\ldots\rangle) \\
&(\texttt{'mary'}, \langle\texttt{'15210'},\texttt{'15213'},\texttt{'15251'},\ldots\rangle), \\
&(\texttt{'peter'}, \langle\texttt{'15150'},\texttt{'15251'},\ldots\rangle), \\
&\ldots \\
\rangle\ &.
\end{aligned}$$

Note that the output sequence is ordered based on the first instance of their key in the input sequences. Similarly, the order of the classes taken by each student are the same as in the input sequence.

9 Aggregation by Iteration

Iteration is a fundamental algorithm technique. It involves a sequence of steps, taken one after another, where each step transforms the state from the previous step. Iteration is an inherently sequential process.

Definition 18.19 (The *iterate* and *iteratePrefixes*)**.** The function *iterate* iterates over a sequence while accumulating a "running sum", i.e., a result that changes at each step. It starts with an initial result and a sequence, and on each step updates the result based on the next element of the sequence.

The function *iterate* has the type signature

$$\mathit{iterate}\ (f : \alpha \times \beta \rightarrow \alpha)\ (x : \alpha)\ (a : \mathbb{S}_\beta) : \alpha$$

where f is a function mapping a state and an element of a to a new state, x is the initial state, a is a sequence.

The semantics of *iterate* is defined as follows.

$$\mathit{iterate}\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ \mathit{iterate}\ f\ (f(x, a[0]))\ (a[1 \cdots |a| - 1]) & \text{otherwise.} \end{cases}$$

A variant of iteration, the function *iteratePrefixes* takes the same arguments as *iterate* but returns a pair, where the first component is a sequence consisting of all the intermediate result computed by iteration, up to and excluding the last element, and the second component is the final results. More precisely, *iteratePrefixes* can be specified as

```
$\mathit{iteratePrefixes}\ f\ x\ a =$
  let $g\ (b, x)\ y = (b +\!\!+\ x, f(x, y))$
  in $\mathit{iterate}\ g\ (\langle\,\rangle, x)\ a$ end
```

Example 18.8. The function *iterate* computes its final result by computing a result for each element of the sequence. Concretely, $\mathit{iterate}\ f\ x\ a$ computes the results x_i, $0 \leq i \leq n = |a|$, where

$$\begin{array}{lcl} x_0 & = & x \\ x_1 & = & f(x_0, a[0]) \\ x_2 & = & f(x_1, a[1]) \\ & \vdots & \\ x_n & = & f(x_{n-1}, a[n-1]). \end{array}$$

The expression

$$\mathit{iterate}\ f\ x\ a$$

thus evaluates to x_n.

The expression

$$\mathit{iteratePrefixes}\ f\ x\ a$$

performs the same computation and returns $(\langle\, x_0, \ldots, x_{n-1}\,\rangle\,, x_n)$.

Example 18.9 (Iteration)**.** For a sequence of length 5, iteration computes its final result as

$$\mathit{iterate}\ f\ x\ a = f(f(f(f(f(v, a[0]), a[1]), a[2]), a[3]), a[4]).$$

For example,

$$\mathit{iterate}\ '+'\ 0\ \langle\, 2, 5, 1, 6\,\rangle$$

returns 14 since it starts with the integer state 0 and then one by one adds the integer elements 2, 5, 1 and 6 of the sequence to the state.

Similarly

$$\mathit{iterate}\ '-'\ 0\ \langle\, 2, 5, 1, 6\, \rangle$$

returns $(((0 - 2) - 5) - 1) - 6 = -14$.

The function

$$\mathit{iterate}\ '+'\ 0\ (\mathit{map}\ \mathit{zeroWhenEven}\ a),$$

which uses the function $\mathit{zeroWhenEven}$ to map even numbers to zero, sums up only the odd numbers in sequence a, returning 6

Exercise 18.1 (Rightmost Positive)**.** Design an algorithm that, for each element in a sequence of integers, finds the rightmost positive number to its left. If there is no positive element to the left of an element, the algorithm returns $-\infty$ for that element.

For example, given the sequence

$$\langle\, 1, 0, -1, 2, 3, 0, -5, 7\, \rangle$$

the algorithm would return

$$\langle\, -\infty, 1, 1, 1, 2, 3, 3, 3\, \rangle\,.$$

Solution. Consider the function

```
$\mathit{extendPositive}\ ((\ell, b), x) =$
  if $x > 0$ then
    $(x, b{+}{+}\langle\, l\, \rangle)$
  else
    $(\ell, b{+}{+}\langle\, \ell\, \rangle)$
```

This function takes as its first argument the tuple consisting of ℓ, the last positive value seen (or $-\infty$) and a sequence b. The second argument x is a new element. The function extends the sequence b with ℓ and returns as the most recently seen positive value x if is positive or ℓ otherwise.

Using this function, we can give an algorithm for the problem of selecting the rightmost positive number to the left of each element is a given sequence a:

```
let $(\ell, b) = \mathit{iterate}\ \mathit{extendPositive}\ (-\infty, \langle\, \rangle)\ a$
in $b$
```

We can solve the same problem more elegantly using $\mathit{iteratePrefixes}$. Consider the function

```
$\mathit{selectPositive}\ (\ell, x) =$
  if $x > 0$ then
    $x$
  else
    $\ell$
```

This function takes as argument ℓ, the last positive value seen, and x, the new element from the sequence. The function then returns x if is positive or ℓ otherwise. We can now we can give an algorithm for the problem of selecting the rightmost positive number preceeding each element in a given sequence a as

```
let $(\ell, b) = \mathit{iteratePrefixes}\ \mathit{selectPositive}\ \ -\infty\ a$
in $b$
```

Note (Iteration and order of operations). Iteration is a powerful technique but can be too big of a hammer, especially when used unnecessarily. For example, when summing the elements in a sequence, we don't need to perform the addition operations in a particular order because addition operations are associative and thus they can be performed in any order desired. The iteration-based algorithm for computing the sum does not take advantage of this property, computing instead the sum in a left-to-right order. As we will see next, we can take advantage of associativity to sum up the elements of a sequence in parallel.

10 Aggregation by Reduction

Reduction. The term ***reduction*** refers to a computation that repeatedly applies an associative binary operation to a collection of elements until the result is reduced to a single value. Recall that associative operations are defined as operations that allow commuting the order of operations.

Associativity.

Definition 18.20 (Assocative Function)**.** A function $f : \alpha \times \alpha \to \alpha$ is associative if $f(f(x, y), z) = f(x, f(y, z))$ for all x, y and z of type α.

Example 18.10. Many functions are associative.

- Addition and multiplication on natural numbers are associative, with 0 and 1 as their identities, respectively.
- Minimum and maximum are also associative with identities ∞ and $-\infty$ respectively.
- The *append* function on sequences is associative, with identity being the empty sequence.
- The union operation on sets is associative, with the empty set as the identity.

Note. Associativity implies that when applying f to some values, the order in which the applications are performed does not matter. Associativity does not mean that you can reorder the arguments to a function (that would be commutativity).

Important (Associativity of Floating Point Operations). Floating point operations are typically not associative, because performing them in different orders can lead to different results because of loss of precision.

Definition 18.21 (The *reduce* operation)**.** In the sequence ADT, we use the function *reduce* to perform a reduction over a sequence by applying an associative binary operation to the elements of the sequence until the result is reduced to a single value. The operation function has the type signature

$$\mathit{reduce}\ (f : \alpha \times \alpha \to \alpha)\ (\mathit{id} : \alpha)\ (a : \mathbb{S}_\alpha) : \alpha$$

where f is an associative function, a is the sequence, and id is the ***left identity*** of f, i.e., $f(\mathit{id}, x) = x$ for all $x \in \alpha$.

When applied to an input sequence with a function f, *reduce* returns the "sum" with respect to f of the input sequence. In fact if f is associative this sum in equal to iteration. We can define the behavior of *reduce* inductively as follows

$$\mathit{reduce}\ f\ \mathit{id}\ a = \begin{cases} \mathit{id} & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\Big(\mathit{reduce}\ f\ \mathit{id}\ (a[0 \cdots \lfloor \frac{|a|}{2} \rfloor - 1]), & \\ \quad \mathit{reduce}\ f\ \mathit{id}\ (a[\lfloor \frac{|a|}{2} \rfloor \cdots |a| - 1]\Big) & \text{otherwise.} \end{cases}$$

Example 18.11 (Reduce and append)**.** The expression

$$\mathit{reduce}\ \mathit{append}\ \langle\ \rangle\ \langle\texttt{'another'},\texttt{'way'},\texttt{'to'},\texttt{'flatten'}\rangle$$

evaluates to

$$\texttt{'anotherwaytoflatten'}.$$

Important. The function *reduce* is more restrictive than *iterate* because it is the same function but with extra restrictions on its input (i.e. that f be associative, and id is a left identity). If the function f is associative, then we have

$$\mathit{reduce}\ f\ \mathit{id}\ a = \mathit{iterate}\ f\ \mathit{id}\ a.$$

Exercise 18.2. Give an example function f, a left identity x, and an input sequence a such that $\mathit{iterate}\ f\ x\ a$ and $\mathit{reduce}\ f\ x\ a$ return different results.

Important. Although we will use *reduce* only with associative functions, we define it for all well-typed functions. To deal properly with functions that are non-associative, the specification of *reduce* makes precise the order in which the argument function f is applied. For instance, when reducing with floating point addition or multiplication, we will need to take the order of operations into account. Because the specification defines the order in which the operations are applied, every (correct) implementation of *reduce* must return the same result: the result is deterministic regardless of the specifics of the algorithm used in the implementation.

Exercise 18.3. Given that *reduce* and *iterate* are equivalent for assocative functions, why would we use *reduce*?

Solution. Even though the input-output behavior of *reduce* and *iterate* may match, their cost specifications differ: unlike *iterate*, which is strictly sequential, *reduce* is parallel. In fact, as we will see in this Chapter , the span of *iterate* is linear in the size of the input, whereas the span of *reduce* is logarithmic.

11 Aggregation with Scan

The $scan$ function. When we restrict ourselves to associative functions, the input-output behavior of the function $reduce$ can be defined in terms of the $iterate$. But the reverse is not true: $iterate$ cannot always be defined in terms of $reduce$, because $iterate$ can use the results of intermediate states computed on the prefixes of the sequence, whereas $reduce$ cannot because such intermediate states are not available. We now describe a function called $scan$ that allows using the results of intermediate computations and also does so in parallel.

Definition 18.22 (The functions $scan$ and $iScan$)**.** The term "scan" refers to a computation that reduces every prefix of a given sequence by repeatedly applying an associative binary operation. The $scan$ function has the type signature

$$scan\ (f : \alpha * \alpha \to \alpha)\ (id : \alpha)\ (a : \mathbb{S}_\alpha)\ :\ (\mathbb{S}_\alpha * \alpha),$$

where f is an associative function, a is the sequence, and id is the left identity element of f.

The expression $scan\ f\ a$ evaluates to the cumulative "sum" with respect to f of all prefixes of the sequence a. For this reason, the $scan$ function is referred to as ***prefix sums***.

We specify the semantics of $scan$ in terms of $reduce$ as follows.

$$\begin{aligned} scan\ f\ id\ a\ =\ & (\langle\, reduce\ f\ id\ a[0 \cdots (i-1)] : 0 \le i < |a|\,\rangle\,, \\ & reduce\ f\ id\ a) \end{aligned}$$

For the definition, we assume that $a[0 \cdots -1] = \langle\ \rangle$.

When computing the result for position i, $scan$ does not include the element of the input sequence at that position. It is sometimes useful to do so. To this end, we define $scanI$ ("I" stands for "inclusive").

We define the semantics of $scanI$ in terms of $reduce$ as follows.

$$scanI\ f\ id\ a\ =\ \langle\, reduce\ f\ id\ a[0 \cdots i] : 0 \le i < |a|\,\rangle$$

Example 18.12 (Scan)**.** Consider the sequence $a = \langle\, 0, 1, 2\,\rangle$. The prefixes of a are

- $\langle\ \rangle$
- $\langle\, 0\,\rangle$
- $\langle\, 0, 1\,\rangle$
- $\langle\, 0, 1, 2\,\rangle$.

The prefixes of a sequence are all the subsequences of the sequence that starts at its beginning. Empty sequence is a prefix of any sequence. The computation $scan$ `'+' $0\ \langle\, 0, 1, 2\,\rangle$

can be written as

$$
\begin{aligned}
\mathit{scan}~\text{`}{+}\text{`}~0~\langle 0,1,2\rangle =& (\,\langle~~\mathit{reduce}~\text{`}{+}\text{`}~0~\langle~\rangle\,, \\
&\qquad\quad \mathit{reduce}~\text{`}{+}\text{`}~0~\langle 0\rangle\,, \\
&\qquad\quad \mathit{reduce}~\text{`}{+}\text{`}~0~\langle 0,1\rangle \\
&\qquad\ \ \rangle~, \\
&\qquad \mathit{reduce}~\text{`}{+}\text{`}~0~\langle 0,1,2\rangle \\
&\quad) \\
=& (\,\langle 0,0,1\rangle\,,3\,)\,.
\end{aligned}
$$

The computation $\mathit{scanI}~\text{`}{+}\text{`}~0~\langle 0,1,2\rangle$ can be written as

$$
\begin{aligned}
\mathit{scanI}~\text{`}{+}\text{`}~0~\langle 0,1,2\rangle ~=~& \langle~~\mathit{reduce}~\text{`}{+}\text{`}~0~\langle 0\rangle\,, \\
&\quad \mathit{reduce}~\text{`}{+}\text{`}~0~\langle 0,1\rangle\,, \\
&\quad \mathit{reduce}~\text{`}{+}\text{`}~0~\langle 0,1,2,\,\rangle \\
&\ \rangle \\
=~& \langle 0,1,3\rangle\,.
\end{aligned}
$$

Note (Scan versus reduce). Since scan can be specified in terms of reduce, one might be tempted to argue that it is redundant. In fact, it is not: as we shall see, performing reduce repeatedly on every prefix is not work efficient. Remarkably scan can be implemented by performing essentially the same work and span of reduce.

Example 18.13 (Copy scan)**.** Scan is useful when we want pass information along the sequence. For example, suppose that we are given a sequence of type $\mathbb{S}_{\mathbb{N}}$ consisting only of integers and asked to return a sequence of the same length where each element receives the previous positive value if any and $-\infty$ otherwise. For the example, for input $\langle\, 0,~7,~0,~0,~3,~0\,\rangle$, the result should be $\langle -\infty, -\infty,~7,~7,~7,~3\,\rangle$.

We considered this problem in an example before and presented an algorithm based on iteration. Because that algorithm uses iteration, it is sequential. But does it have to be sequential? There is, perhaps, a parallel algorithm.

We can solve this problem using an inclusive scan, if we can find with a combining function f that does the crux of the work. Consider the function

$$\mathit{selectPositive}~(x, y) = \texttt{if}~y > 0~\texttt{then}~y~\texttt{else}~x.$$

The function returns its right (second) argument if it is positive, otherwise it returns its the left (first) argument.

To be used in a scan, $\mathit{selectPositive}$ must be associative. That is, for all x, y and z, the following equality should hold:

$$\mathit{selectPositive}(x, \mathit{selectPositive}(y, z)) = \mathit{selectPositive}(\mathit{selectPositive}(x, y), z).$$

There are eight possibilities corresponding to the signs of x, y and z. When $z > 0$, the left and right hand sides of the equality both yield z. When $z \leq 0$ and $y > 0$, the left and right hand sides both yield y. Finally, when $z \leq 0$ and $y \leq 0$, they left and right hand sides both yield x.

To use $\mathit{selectPositive}$ in a scan, we also need its left identity. Because

$$\mathit{selectPositive}\ (-\infty, y) = y$$

for any y, the left identity for $\mathit{selectPositive}$ is $-\infty$.

Remark (Reduce and scan). Experience in parallel computing shows that reduce and scan are powerful primitives that suffice to express many parallel algorithms on sequences. In some ways this is not surprising, because the functions allow using two important algorithm-design techniques: reduce function allows expressing divide-and-conquer algorithms and the scan function allows expressing iterative algorithms.