Lecture 8 — Sets and Tables II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 20, 2012

Today:

- Finish Tables (from last class)
- Example of Tables for indexing the web
- Single Threaded Sequences

1 Building and searching an index

Here we consider an application of sets and tables. In particular, the goal is to generate an index of the sort that Google or Bing create so that a user can make word queries and find the documents in which those words occur. We will consider logical queries on words involving `And`, `Or`, and `AndNot`. For example a query might look like

"CMU" `And` "fun" `And` ("courses" `Or` "clubs")

and it would return a list of web pages that match the query (*i.e.*, contain the words "CMU", "fun" and either "courses" or "clubs"). This list would include the 15-210 home page, of course.

These kinds of searchable indexes date back to the 1970s with systems such as Lexis for searching law documents. Today, beyond web searches, searchable indices are an integral part of most mailers and operating systems. The different indices support somewhat different types of queries. For example, by default Google supports queries with `And` and `adjacent to` but with their advanced search you can search with `Or`, `AndNot` as well as other types of searches.

Let's imagine we want to support the following interface

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index
  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
end
```

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

The input to `makeIndex` is a sequence of pairs each consisting of a document identifier (e.g. the URL) and the contents of the document as a single text string. So for example we might want to index recent tweets, that might include the following "documents":

$T = \langle$ ("jack", "chess club was fun"),
("mary", "I had a fun time in 210 class today"),
("nick", "food at the cafeteria sucks"),
("sue", "In 217 class today I had fun reading my email"),
("peter", "I had fun at nick's party"),
("john", "tiddlywinks club was no fun, but more fun than 218"),
$\rangle$

where the identifiers are the names, and the contents is the tweet.

The interface can be used to make an index of these tweets:

val $f = (\texttt{find}\ (\texttt{makeIndex}(T))) : \texttt{word} \to \texttt{doclist}$

In addition to making the index, this partially applies `find` on it. We can then use this index for various queries. For example:

```
 toSeq(And(f "fun", Or(f "class", f "club")))
⇒ ⟨"jack", "mary", "sue", "john"⟩
```

returns all the documents (tweets) that contain "fun" and either "class" or "club", and

```
 size(AndNot(f "fun", f "tiddlywinks"))
⇒ 4
```

returns the number of documents that contain "fun" and not "tiddlywinks".

We can implement this interface very easily using sets and tables. The `makeIndex` function can be implemented as follows.

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id,str) = ⟨(w,id) : w ∈ tokens(str)⟩
4    val Pairs = flatten ⟨ tagWords(d) : d ∈ docs ⟩
5    val Words = Table.collect(Pairs)
6  in
7    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8  end
```

The `tagWords` function takes a document as a pair consisting of the document identifier and contents, breaks the string into tokens (words) and tags each token with the identifier returning a sequence of these pairs. For example, on the document

```
 tagWords("jack", "chess club was fun")
⇒ ⟨ ("chess","jack"),("club", "jack"), ("was", "jack"), ("fun", "jack") ⟩
```

The function `tagWords` is then applied to all document and the result flattened so it is a single sequence. In our example the result would start as:

$$\begin{aligned}\texttt{Pairs} \;=\; \langle &(\text{“chess”},\text{“jack”}),(\text{“club”},\text{“jack”}),\ (\text{“was”},\text{“jack”}),\\ &(\text{“fun”},\text{“jack”}),\ (\text{“I”},\text{“mary”}),\ (\text{“had”},\text{“mary”}),\ (\text{“fun”},\text{“mary”}),\ldots\end{aligned}$$

The `Table.collect` then collects the entries by word creating a sequence of matching documents. In our example it would start:

$$\begin{aligned}\texttt{Words} \;=\; \langle &(\text{“a”},\ \langle \text{“mary”}\rangle),\\ &(\text{“at”},\ \langle \text{“mary”},\text{“peter”}\rangle),\\ &\ldots\\ &(\text{“fun”},\ \langle \text{“jack”},\text{“mary”},\text{“sue”},\text{“peter”},\text{“john”}\rangle),\\ &\ldots\end{aligned}$$

Finally, for each word the sequences of document identifiers is converted to a set. Note the notation that is used to express a map over the elements of a table.

Assuming that all tokens have a length upper bounded by a constant, the cost of `makeIndex` is dominated by the collect, which is basically a sort. The work is therefore $O(n \log n)$ and the span is $O(\log^2 n)$, assuming the words have constant length. The rest of the interface can be implemented as follows:

fun `find` $T\ v = $ `Table.find` $T\ v$

fun `And`$(s_1, s_2) = s_1 \cap s_2$

fun `Or`$(s_1, s_2) = s_1 \cup s_2$

fun `AndNot`$(s_1, s_2) = s_1 \setminus s_2$

fun `size`$(s) = |s|$

fun `toSeq`$(s) =$ `Set.toSeq`(s)

Note that if we do a `size(f "red")` the cost is only $O(\log n)$ work and span. It just involves a search and then a length.

If we do `And(f "fun", Or(f "courses", f "classes"))` the worst case work and span are at most:

$$\begin{aligned} W &= O(|f(\text{"}fun\text{"})| + |f(\text{"}courses\text{"})| + |f(\text{"}classes\text{"})|)\\ S &= O(\log |index|)\end{aligned}$$

The sum of sizes is to account for the cost of the `And` and `Or`. The actual cost could be significantly less especially if one of the sets is very small.

2 Single-Threaded Array Sequences

In this course we will be using purely functional code because it is safe for parallelism, easier to reason about formally, and because its cool. For many algorithms using the purely functional version makes no difference in the asymptotic work bounds—for example quickSort and mergeSort use $\Theta(n \log n)$ work (expected case for quickSort) whether purely functional or imperative. However, in some cases purely functional implementations lead to up to a $O(\log n)$ factor of additional work. To avoid this we will slightly cheat in this class and allow for benign "effect" under the hood in exactly one ADT, described in this section. These effects do not affect the observable values (you can't observe them by looking at results), but they do affect cost analysis—and if you sneak a peak at our implementation, you will see some side effects.

The issue has to do with updating positions in a sequence. In an imperative language updating a single position can be done in "constant time". In the functional setting we are not allowed to change the existing sequence, everything is persistent. This means that for a sequence of length n an update can either be done in $O(n)$ work with an arraySequence (the whole sequence has to be copied before the update) or $O(\log n)$ work with a treeSequence (an update involves traversing the path of a tree to a leaf). In fact you might have noticed that our sequence interface does not even supply a function for updating a single position. The reason is both to discourage sequential computation, but also because it would be expensive.

Consider a function `update` (i, v) S that updates sequence S at location i with value v returning the new sequence. This function would have cost $O(|S|)$ in the arraySequence cost specification. Someone might be tempted to write a sequential loop using this function. For example for a function $f : \alpha -> \alpha$, a `map` function can be implemented as follows:

```
fun map f S =
  iter (fn ((i,S'),v) => (i+1, update (i, f(v)) S'))
       (0,S)
       S
```

This code iterates over S with i going from 0 to $n-1$ and at each position i updates the value S_i with $f(S_i)$. The problem with this code is that even if f has constant work, with an `arraySequence` this will do $O(|S|^2)$ total work since every update will do $O(|S|)$ work. By using a `treeSequence` implementation we can reduce the work to $O(|S| \log |S|)$ but that is still a factor of $O(\log |S|)$ off of what we would like.

In the class we sometimes do need to update either a single element or a small number of elements of a sequence. We therefore introduce an ADT we refer to as a *Single Threaded Sequence* (`stseq`). Although the interface for this ADT is quite straightforward, the cost specification is somewhat tricky. To define the cost specification we need to distinguish between the latest "copy" of an instance of an `stseq`, and earlier copies. Basically whenever we update a sequence we create a new "copy", and the old "copy" is still around due to the persistence in functional languages. The cost specification is going to give different costs for updating the latest copy and old copies. Here we will only define the cost for updating the latest copy, since this is the only way we will be using an `stseq`. The interface and costs is as follows:

	Work	Span
`fromSeq(S) : α seq → α stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST) : α stseq → α seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST` i `: α stseq → int → α` Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i,v) S `: (int × α) → α stseq → α stseq` Replaces the i^{th} element of S with v.	$O(1)$	$O(1)$
`inject` I S `: (int × α) seq → α stseq → α stseq` For each $(i,v) \in I$ replaces the i^{th} element of S with v.	$O(\|I\|)$	$O(1)$

An `stseq` is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (`nth`), update a position of the sequence (`update`) or update multiple positions in the sequence (`inject`). To use other functions from the sequence library, one needs to covert an `stseq` back to a sequence (using `toSeq`).

In the cost specification the work for both `nth` and `update` is $O(1)$, which is about as good as we can get. Again, however, this is only when S is the latest version of a sequence (i.e. noone else has updated it). The work for `inject` is proportional to the number of updates. It can be viewed as a parallel version of `update`.

Now with an `stseq` we can implement our map as follows:

```
1  fun map f S = let
2    val S' = StSeq.fromSeq(S)
3    val R = iter (fn ((i,S''),v) ⇒ (i+1, StSeq.update (i,f(v)) S''))
4                 (0,S')
5                 S
6  in
7    StSeq.toSeq(R)
8  end
```

This implementation first converts the input sequence to an `stseq`, then updates each element of the `stseq`, and finally converts back to a sequence. Since each update takes constant work, and assuming the function f takes constant work, the overall work is $O(n)$. The span is also $O(n)$ since `iter` is completely sequential. This is therefore not a good way to implement `map` but it does illustrate that the work of multiple updates can be reduced from $O(n^2)$ on array sequences or $O(n \log n)$ on tree sequences to $O(n)$ using an `stseq`.

Implementing Single Threaded Sequences. You might be curious about how single threaded sequences can be implemented so they act purely functional but match the cost specification. Here we will just briefly outline the idea.

The trick is to keep two copies of the sequence (the original and the current copy) and additionally to keep a "change log". The change log is a linked list storing all the updates made to the original sequence. When converting from a sequence to an `stseq` the sequence is copied to make a second identical copy (the current copy), and an empty change log is created. A different representation is now used for the latest version and old versions of an `stseq`. In the latest version we keep both copies (original and current) as well as the change log. In the old versions we only keep the original copy and the change log. Lets consider what is needed to update either the current or an old version. To update the current version we modify the current copy in place with a side effect (non functionally), and add the change to the change log. We also take the previous version and mark it as an old version removing its current copy. When updating an old version we just add the update to its change log. Updating the current version requires side effects since it needs to update the current copy in place, and also has to modify the old version to mark it as old and remove its current copy.

Either updating the current version or an old version takes constant work. The problem is the cost of `nth`. When operating on the current version we can just look up the value in the current copy, which is up to date. When operating on an old version, however, we have to go back to the original copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

In this course we will use `stseq`s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

3 SML Code

3.1 Indexes

```
functor TableIndex(Table : TABLE where type Key.t = string) : INDEX =
struct

structure Seq = Table.Seq
structure Set = Table.Set

type word = string
type docId = string
type 'a seq = 'a Seq.seq
type docList = Table.set
type index = docList Table.table

fun makeIndex docs =
let
  fun toWords str = Seq.tokens (fn c => not (Char.isAlphaNum c)) str

  fun tagWords(docId,str) = Seq.map (fn t => (t, docId)) (toWords str)

  (* generate all word-documentid pairs *)
  val allPairs = Seq.flatten (Seq.map tagWords docs)
```

```
  (* collect them by word *)
  val wordTable = Table.collect allPairs

in
  (* convert the sequence of documents for each word into a set
     which removes duplicates*)
  Table.map Set.fromSeq wordTable
end

fun find Idx w =
   case (Table.find Idx w) of
      NONE => Set.empty
    | SOME(s) => s

val And = Set.intersection
val AndNot = Set.difference
val Or = Set.union
val size = Set.size
val toSeq = Set.toSeq

end
```