Functional Data Structures

Mermer Dupree

1 Functional Basics

1.1 Benefits [3]

- Function have no side effects
  - This makes it ideal for higher order functions
  - Prevents many bugs from occurring
  - functions can be tested more thoroughly
  - Lazy evaluation can be used automatically eliminate unnecessary tasks
- Makes concurrency easier.
  - No race conditions
  - Processors aren't getting much faster.
  - Ultra parallel machines with possibly hundreds of cores are in the near future
  - Hardware makers are considering removing caches from processors making cache friendliness less important
  - Dataflow can be analyzed to automatically distribute tasks over many machines and even be optimized for data locality
- Usually less lines of code, but each line takes more thinking to write
- Memory hungry but memory has become cheap
- Easier to prove program's correctness
  - Functions often mirror mathematical proofs

1.2 Monoids [1]

- A data type and an operation defined on that data type that is: closed, associative and has an identity element
- Benefits of Closed property
  - Operator can be chained (e.g., $1 + 2 + 3 + 4$). There for operator can be used to reduce a set

- Benefits of Associative property
  - Divide and conquer algorithms
    - Basis for well know aggregation algorithm
  - Parallelization
    - Divide and Conquer algorithms can easily be parallelized
  - Incrementalism
    - For example if you calculated the sum of 1 to 10, if you want to next calculate the sum from 1 to 11 you don't have to start from scratch you can simply add 11 to the sum you've already calculated
  - Benefits of the identity element
    - This feature is less important, but does allow you perform reduce on an empty list and makes it so an initial value for reduce does not need to be specified

1.3 Monads

- a monad is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together. This allows the programmer to build pipelines that process data in a series of steps (also called actions), in which each action is decorated with additional processing rules provided by the monad. [8]
- a monad is an object with two functions **unit** (or return in Haskell) and **bind** (`>>=` in Haskell)
  - **unit** (unit(value) $\rightarrow$ monad) is a function that takes a value and returns a monad. This is a constructor of the monad
  - **bind** (bind(monad, function(value) $\rightarrow$ monad) $\rightarrow$ monad) returns a monad and takes 2 arguments: a monad and a function that takes a value and return a monad
    - for the following examples `a` is a plain value, `M` is an instance of a monad and `func`, `func1`,`func2`, ... are functions that take a value and return a monad
    - bind can be written in a more object oriented notation. Instead of `bind(M,func)` we can write `M.bind(func)`
    - The bind operator unwraps the plain value `a` embedded in its input monadic value `M`, and feeds it to the function. [8]

```
unit(a).bind(func) == func(a)

M.bind(unit) == M
```

    - The function then creates a new monadic value that can be fed to the next bind operators composed in the pipeline. [8]

```
M.bind(func1).bind(func2).bind(func3)
```

- the monad type constructor defines a type of computation, the `return` function creates primitive values of that computation type and `>>=` combines computations of that type together to make more complex computations of that type.
- Using a container analogy, the type constructor `m` is a container that can hold different values. `m a` is a container holding a value of type `a`. The return function puts a value into a monad container. The `>>=` function takes the value from a monad container and passes it to a function to produce a monad container containing a new value, possibly of a different type.

1.4 Algebraic Data types

- The values of a product type typically contain several values, called fields. All values of that type have the same combination of field types. The set of all possible values of a product type is the set-theoretical product, i.e. the Cartesian product, of the sets of all possible values of its field types
- The values of a sum type are typically grouped into several classes, called variants. The set of all possible values of a sum type is the set-theoretical sum, i.e. the disjoint union, of the sets of all possible values of its variants. An Enumerated type is an example of a sum type.

2 Parallelism

2.1 Implicitly Parallel Languages

- SAC (Single Assignment C) - is a strict purely functional programming language whose design is focused on the needs of numerical applications
- SISAL ("Streams and Iteration in a Single Assignment Language") - is a general-purpose single assignment functional programming language with strict semantics, implicit parallelism, and efficient array handling.
- ZPL (Z-level Programming Language) is an array programming language designed to replace C and C++ programming languages in engineering and scientific applications
- lattice data structure used for automating parallelism in Swift-T function HPC scripting language [10]

3 Functional Data Structures [2]

3.1 Linked List

- Description
  - a LIFO data structure
- Implementation
  - A list of nodes which consist of a value and a pointer to the next node

- Complexity

Complexity	
First	$O(1)$
Last	$O(n)$
Nth	$O(n)$
Prepend	$O(1)$
Append	$O(n)$
Insert	$O(n)$
Concat	$O(n)$

- Memory and Locality
  - Prepend does not require copying the list. Append does and Insert requires partial coping
  - Not cache friendly, but language features exist to make sure elements in the list are close to each other in the heap

3.2 Banker's Queue

- Description
  - A FIFO queue.
- Implementation
  - Two lazy singly-linked list: front list for dequeuing and rear list for enqueuing.
  - When the front list is larger than the rear list the rear list reversed and concatenated to the front list.
- Complexity

Complexity	
First	$O(1)$
Last	$O(1)$*
Nth	$O(n)$
Prepend	$O(1)$
Append	$O(n)$
Insert	$O(n)$
Concat	$O(n)$

- Memory and Locality
  - The Last complexity is amortized. It has to perform expensive copying sometimes, but it becomes exponentially infrequent over each operation
  - Not cache friendly, but language features exist to make sure elements in the list are close to each other in the heap

3.3 2-3 Finger Tree

- Description

- Flexible data structure. May be used to implement efficient random-access sequences, ordered sequences, interval trees, and priority queues

- Complexity

Complexity	
First	$O(1)$
Last	$O(1)$
Nth	$O(\log n)$
Prepend	$O(1)$
Append	$O(1)$
Insert	$O(\log n)$
Concat	$O(n)$

- Memory and Locality
  - Slow in practice because it is not cache friendly at all, and the language features can't help it be cache friendly

3.4 Red Black Tree

- Description
  - Main associative data structure in functional programing
  - Balanced binary search tree
- Implementation
  - every path from root to leaf contains the same number of black nodes
  - no red node has a red parent
  - must rebalance after an update
- Complexity

Complexity	
Get	$O(\log n)$
Insert	$O(\log n)$
Update	$O(\log n)$
Intersect	$O(n)$
Union	$O(n)$

3.5 Bitmapped Vector Tri

- Description
  - both associative and sequential data structure
  - Provides fast access to all parts: head tail and anything in the middle
  - known as a vector in clojure

- Implementation
  - start with array copy on write up until it reaches size 32
  - when it reaches the max size we put it in an array of arrays up to size 32
  - once all 32 arrays are full then you put them in an array of arrays of arrays
  - max depth would be 7
- Complexity

Complexity	
First	$O(1)$
Last	$O(1)$
Nth	$O(1)$
Prepend	$O(n)$
Append	$O(1)$
Insert	$O(n)$
Concat	$O(n)$
Update	$O(1)$

  - not quite constant. (Its actually $O(\log_3 2n)$) but for all practical purposes (any dataset that fits in memory) it is constant.
- Memory and Locality
  - Very fast in practice. Cache friendly.

3.6 Brodal Tree [9]

- Description
  - Functional Priority Queue
- Implementation
  - A heap-ordered tree
  - if a node is of node is of rank k then its children are of rank $j < k$ (rank is height)
  - if a rank k node has a child or rank j then it has between 2 and 7 children of rank j
  - a node of rank k always has at least 2 children of rank $k - 1$
  - so a node will always have at most $O(\log n)$ children because a node of rank k always has at least $2^{k+1}1$ nodes, so no node can have rank more than $O(\log n)$.

References

[1] Monoids: `http://fsharpforfunandprofit.com/posts/monoids-without-tears/`

[2] Daniel Spiewak - Extreme Cleverness: Functional Data Structures in Scala: `https://www.youtube.com/watch?v=pNhBQJN44YQ`

[3] Robert C Martin - Functional Programming: What? Why? When? https://www.youtube.com/watch?v=7Zlp9rKHGD4

[4] Scala : https://learnxinyminutes.com/docs/scala/

[5] Types in F# : http://fsharpforfunandprofit.com/posts/type-abbreviations/

[6] Monads in F# : http://santialbo.com/blog/2013/03/27/monads-in-f-sharp/

[7] Haskell All About Monads : https://wiki.haskell.org/All_About_Monads

[8] Mondad Wikipedia : https://en.wikipedia.org/wiki/Monad_(functional_programming)

[9] How do Brodal queues work? : https://www.quora.com/How-do-Brodal-queues-work

[10] LVars: Lattice-based Data Structures for Deterministic Parallelism, Lindsey Kuper, Ryan R. Newton