Persistent data structure

In computing, a **persistent data structure** or **not ephemeral data structure** is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. The term was introduced in Driscoll, Sarnak, Sleator, and Tarjans' 1986 article.[1]

A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. The data structure is **fully persistent** if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called **confluently persistent**. Structures that are not persistent are called *ephemeral*.[2]

These types of data structures are particularly common in logical and functional programming,[2] as languages in those paradigms discourage (or fully forbid) the use of mutable data.

Contents

Partial versus full persistence

In the partial persistence model, a programmer may query any previous version of a data structure, but may only update the latest version. This implies a linear ordering among each version of the data structure.[3] In the fully persistent model, both updates and queries are allowed on any version of the data structure. In some cases the performance characteristics of querying or updating older versions of a data structure may be allowed to degrade, as is true with the Rope data structure.[4] In addition, a data structure can be referred to as confluently persistent if, in addition to being fully persistent, two versions of the same data structure can be combined to form a new version which is still fully persistent.[5]

Techniques for preserving previous versions

Copy-on-write

One method for creating a persistent data structure is to use a platform provided ephemeral data structure such as an array to store the data in the data structure and copy the entirety of that data structure using copy-on-write semantics for any updates to the data structure. This is an inefficient technique because the entire backing data structure must be copied for each write, leading to worst case O(n·m) performance characteristics for m modifications of an array of size n.

Fat node

The fat node method is to record all changes made to node fields in the nodes themselves, without erasing old values of the fields. This requires that nodes be allowed to become arbitrarily “fat”. In other words, each fat node contains the same information and pointer fields as an ephemeral node, along with space for an arbitrary number of extra field values. Each extra field value has an associated field name and a version stamp which indicates the version in which the named field was changed to have the specified value. Besides, each fat node has its own version stamp, indicating the version in which the node was created. The only purpose of nodes having version stamps is to make sure that each node only contains one value per field name per version. In order to navigate through the structure, each original field value in a node has a version stamp of zero.

Complexity of fat node

With using fat node method, it requires O(1) space for every modification: just store the new data. Each modification takes O(1) additional time to store the modification at the end of the modification history. This is an amortized time bound, assuming modification history is stored in a growable array. At access time, the right version at each node must be found as the structure is traversed. If "m" modifications were to be made, then each access operation would have O(log m) slowdown resulting from the cost of finding the nearest modification in the array.

Path copying

With the path copying method a copy of all nodes is made on the path to any node which is about to be modified. These changes must then be cascaded back through the data structure: all nodes that pointed to the old node must be modified to point to the new node instead. These modifications cause more cascading changes, and so on, until the root node is reached.

Complexity of path copying

With m modifications, this costs O(log m) additive lookup time. Modification time and space are bounded by the size of the longest path in the data structure and the cost of the update in the ephemeral data structure. In a Balanced Binary Search Tree without parent pointers the worst case modification time complexity is O(log n + update cost). However, in a linked list the worst case modification time complexity is O(n + update cost).

A combination

Driscoll, Sarnak, Sleator, Tarjan came up[1] with a way to combine the techniques of fat nodes and path copying, achieving O(1) access slowdown and O(1) modification space and time complexity.

In each node, one modification box is stored. This box can hold one modification to the node—either a modification to one of the pointers, or to the node's key, or to some other piece of node-specific data—and a timestamp for when that modification was applied. Initially, every node's modification box is empty.

Whenever a node is accessed, the modification box is checked, and its timestamp is compared against the access time. (The access time specifies the version of the data structure being considered.) If the modification box is empty, or the access time is before the modification time, then the modification box is ignored and only the normal part of the node is considered. On the other hand, if the access time is after the modification time, then the value in the modification box is used, overriding that value in the node.

Modifying a node works like this. (It is assumed that each modification touches one pointer or similar field.) If the node's modification box is empty, then it is filled with the modification. Otherwise, the modification box is full. A copy of the node is made, but using only the latest values. The modification is performed directly on the new node, without using the modification box. (One of the new node's fields overwritten and its modification box stays empty.) Finally, this change is cascaded to the node's parent, just like path copying. (This may involve filling the parent's modification box, or making a copy of the parent recursively. If the node has no parent—it's the root—it is added the new root to a sorted array of roots.)

With this algorithm, given any time t, at most one modification box exists in the data structure with time t. Thus, a modification at time t splits the tree into three parts: one part contains the data from before time t, one part contains the data from after time t, and one part was unaffected by the modification.

Complexity of the combination

Time and space for modifications require amortized analysis. A modification takes O(1) amortized space, and O(1) amortized time. To see why, use a potential function ϕ, where ϕ(T) is the number of full live nodes in T . The live nodes of T are just the nodes that are reachable from the current root at the current time (that is, after the last modification). The full live nodes are the live nodes whose modification boxes are full.

Each modification involves some number of copies, say k, followed by 1 change to a modification box. Consider each of the k copies. Each costs O(1) space and time, but decreases the potential function by one. (First, the node to be copied must be full and live, so it contributes to the potential function. The potential function will only drop, however, if the old node isn't reachable in the new tree. But it is known that it isn't reachable in the new tree—the next step in the algorithm will be to modify the node's parent to point at the copy. Finally, it is known that the copy's modification box is empty. Thus, replaced a full live node has been replaced with an empty live node, and ϕ goes down by one.) The final step fills a modification box, which costs O(1) time and increases ϕ by one.

Putting it all together, the change in ϕ is $\Delta\phi = 1 - k$. Thus, the algorithm takes $O(k + \Delta\phi) = O(1)$ space and $O(k + \Delta\phi + 1) = O(1)$ time

Generalized form of persistence

Path copying is one of the simple methods to achieve persistency in a certain data structure such as binary search trees. It is nice to have a general strategy for implementing persistence that works with any given data structure. In order to achieve that, we consider a directed graph G. We assume that each vertex v in G has a constant number c of outgoing edges that are represented by pointers. Each vertex has a label representing the data. We consider that a vertex has a bounded number d of edges leading into it which we define as inedges(v). We allow the following different operations on G.

- CREATE-NODE(): Creates a new vertex with no incoming or outgoing edges.
- CHANGE-EDGE(v, i, u): Changes the i^{th} edge of v to point to u
- CHANGE-LABEL(v, x): Changes the value of the data stored at v to x

Any of the above operations is performed at a specific time and the purpose of the persistent graph representation is to be able to access any version of G at any given time. For this purpose we define a table for each vertex v in G. The table contains c columns and $d + 1$ rows. Each row contains in addition to the pointers for the outgoing edges, a label which represents the data at the vertex and a time t at which the operation was performed. In addition to that there is an array inedges(v) that keeps track of all the incoming edges to v. When a table is full, a new table with $d + 1$ rows can be created. The old table becomes inactive and the new table becomes the active table.

CREATE-NODE

A call to CREATE-NODE creates a new table and set all the references to null

CHANGE-EDGE

If we assume that CHANGE-EDGE(v, i, u) is called, then there are two cases to consider.

- There is an empty row in the table of the vertex v: In this case we copy the last row in the table and we change the ith edge of vertex v to point to the new vertex u

- Table of the vertex *v* is full: In this case we need to create a new table. We copy the last row of the old table into the new table. We need to loop in the array inedges(*v*) in order to let each vertex in the array point to the new table created. In addition to that, we need to change the entry *v* in the inedges(w) for every vertex *w* such that edge *v,w* exists in the graph G.

CHANGE-LABEL

It works exactly the same as CHANGE-EDGE except that instead of changing the i^{th} edge of the vertex, we change the label.

Efficiency of the generalized persistent data structure

In order to find the efficiency of the scheme proposed above, we use an argument defined as a credit scheme. The credit represents a currency. For example the credit can be used to pay for a table. The argument states the following:

- The creation of one table requires one credit
- Each call to CREATE-NODE comes with two credits
- Each call to CHANGE-EDGE comes with one credit

The credit scheme should always satisfy the following invariant: Each row of each active table stores one credit and the table has the same number of credits as the number of rows. Let us confirm that the invariant applies to all the three operations CREATE-NODE, CHANGE-EDGE and CHANGE-LABEL.

- CREATE-NODE: It acquires two credits, one is used to create the table and the other is given to the one row that is added to the table. Thus the invariant is maintained.
- CHANGE-EDGE: There are two cases to consider. The first case occurs when there is still at least one empty row in the table. In this case one credit is used to the newly inserted row. The second case occurs when the table is full. In this case the old table becomes inactive and the $d+1$ credits are transformed to the new table in addition to the one credit acquired from calling the CHANGE-EDGE. So in total we have $d+2$ credits. One credit will be used for the creation of the new table. Another credit will be used for the new row added to the table and the d credits left are used for updating the tables of the other vertices that need to point to the new table. We conclude that the invariant is maintained.
- CHANGE-LABEL: It works exactly the same as CHANGE-EDGE.

As a summary, we conclude that having n_1 calls to CREATE_NODE and n_2 calls to CHANGE_EDGE will result in the creation of $2 \cdot n_1 + n_2$ tables. Since each table has size $O(d)$ then filling in a table requires $O(d^2)$ where the additional d factor comes from updating the inedges at other nodes. Therefore the amount of work required to complete a sequence of operations is bounded by the number of tables created multiplied by $O(d^2)$. Each access operation can be done in $O(Log(d))$ and there are m edge and label operations, thus it requires $m \cdot O(Log(d))$. We conclude that There exists a data structure that can complete any n sequence of CREATE-NODE, CHANGE-EDGE and CHANGE-LABEL in $O(n \cdot d^2) + m \cdot O(Log(d))$.

Applications of persistent data structures

Next Element Search or Point Location

One of the useful applications that can be solved efficiently using persistence is the Next Element Search. Assume that there are n non intersecting line segments that don't cross each other. We want to build a data structure that can query a point p and return the segment above p (if any). We will start by solving the Next Element Search using the naïve method then we will show how to solve it using the persistent data structure method.

Naïve Method

We start with a vertical line segment that starts off at infinity and we sweep the line segments from the left to the right. We take a pause every time we encounter an end point of these segments. The vertical lines split the plane into vertical strips. If there are n line segments then we can get $2 \cdot n + 1$ vertical strips since each segment has 2 end points. No segment begins and ends in the strip. Every segment either it doesn't touch the strip or it completely crosses it. We can think of the segments as some objects that are in some sorted order from top to bottom. What we care about is where the point that we are looking at fits in this order. We sort the endpoints of the segments by their x coordinate. For each strip s_i, we store the subset segments that cross s_i in a dictionary. When the vertical line sweeps the line segments, whenever it passes over the left endpoint of a segment then we add it to the to the dictionary. When it passes through the right endpoint of the segment, we remove it from the dictionary. At every endpoint, we save a copy of the dictionary and we store all the copies sorted by the x coordinates. Thus we have a data structure that can answer any query. In order to find the segment above a point p, we can look at the x coordinate of p to know which copy or strip it belongs to. Then we can look at the y coordinate to find the segment above it. Thus we need two binary searches, one for the x coordinate to find the strip or the copy, and another for the y coordinate to find the segment above it. Thus the query time takes $O(Log(n))$. In this data structure, the space is the issue since if we assume that we have the segments structured in a way such that every segment starts before the end of any other segment then the space required for the structure to be built using the naïve method would be $O(n^2)$. Let us see how we can build another persistent data structure with the same query time but with a better space.

Persistent Data Structure Method

We can notice that what really takes time in the data structure used in the naïve method is that whenever we move from a strip to the next, we need to take a snap shot of whatever data structure we are using to keep things in sorted order. We can notice that once we get the segments that intersect s_i, when we move to s_{i+1} either one thing leaves or one thing enters. If the difference between what is in s_i and what is in s_{i+1} is only one insertion or deletion then it is not a good idea to copy everything from s_i to s_{i+1}. The trick is that since each copy differs from the previous one by only one insertion or deletion, then we need to copy only the parts that change. Let us assume that we have a tree rooted at T. When we insert a key k into the tree, we create a new leaf containing k. Performing rotations to rebalance the tree will only modify the nodes of the path from k to T. Before inserting the key k into the tree, we copy all the nodes on the path from k to T . Now we have 2 versions of the tree, the original one which doesn't contain k and the new tree that contains k and whose root is a copy of the root of T. Since copying the path from k to T doesn't increase the insertion time by more than a constant factor then the insertion in the persistent data structure takes $O(Log(n))$ time. For the deletion, we need to find which nodes will be affected by the deletion. For each node v affected by the deletion, we copy the path from the root to v. This will provide a new tree whose root is a copy of the root of the original tree. Then we perform the deletion on the new tree. We will end up with 2 versions of the tree. The original one which contains k and the new one which doesn't contain k. Since any deletion only modifies the path from the root to v and any appropriate deletion algorithm runs in $O(Log(n))$, thus the deletion in the persistent data structure takes $O(Log(n))$. Every sequence of insertion and deletion will cause the creation of a sequence of dictionaries or versions or trees

$S_1, S_2, \ldots S_i$ where each S_i is the result of operations $S_1, S_2, \ldots S_i$. If each S_i contains m elements, then the search in each S_i takes $O(Log(m))$. Using this persistent data structure we can solve the next element search problem in $O(Log(n))$ query time and $O(n \cdot Log(n))$ space instead of $O(n^2)$.

Examples of persistent data structures

Perhaps the simplest persistent data structure is the singly linked list or *cons*-based list, a simple list of objects formed by each carrying a reference to the next in the list. This is persistent because the *tail* of the list can be taken, meaning the last *k* items for some *k*, and new nodes can be added in front of it. The tail will not be duplicated, instead becoming shared between both the old list and the new list. So long as the contents of the tail are immutable, this sharing will be invisible to the program.

Many common reference-based data structures, such as red–black trees,[6] stacks,[7] and treaps,[8] can easily be adapted to create a persistent version. Some others need slightly more effort, for example: queues, dequeues, and extensions including min-deques (which have an additional *O*(1) operation *min* returning the minimal element) and random access deques (which have an additional operation of random access with sub-linear, most often logarithmic, complexity).

There also exist persistent data structures which use destructive operations, making them impossible to implement efficiently in purely functional languages (like Haskell outside specialized monads like state or IO), but possible in languages like C or Java. These types of data structures can often be avoided with a different design. One primary advantage to using purely persistent data structures is that they often behave better in multi-threaded environments.

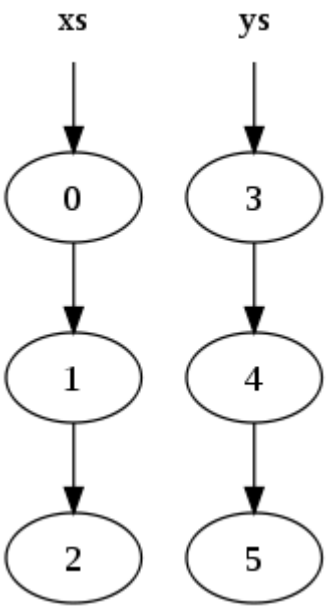
Linked lists

Singly linked lists are the bread-and-butter data structure in functional languages.[9] Some ML-derived languages, like Haskell, are purely functional because once a node in the list has been allocated, it cannot be modified, only copied, referenced or destroyed by the garbage collector when nothing refers to it. (Note that ML itself is **not** purely functional, but supports non-destructive list operations subset, that is also true in the Lisp (LISt Processing) functional language dialects like Scheme and Racket.)

Consider the two lists:

```
xs = [0, 1, 2]
ys = [3, 4, 5]
```

These would be represented in memory by:

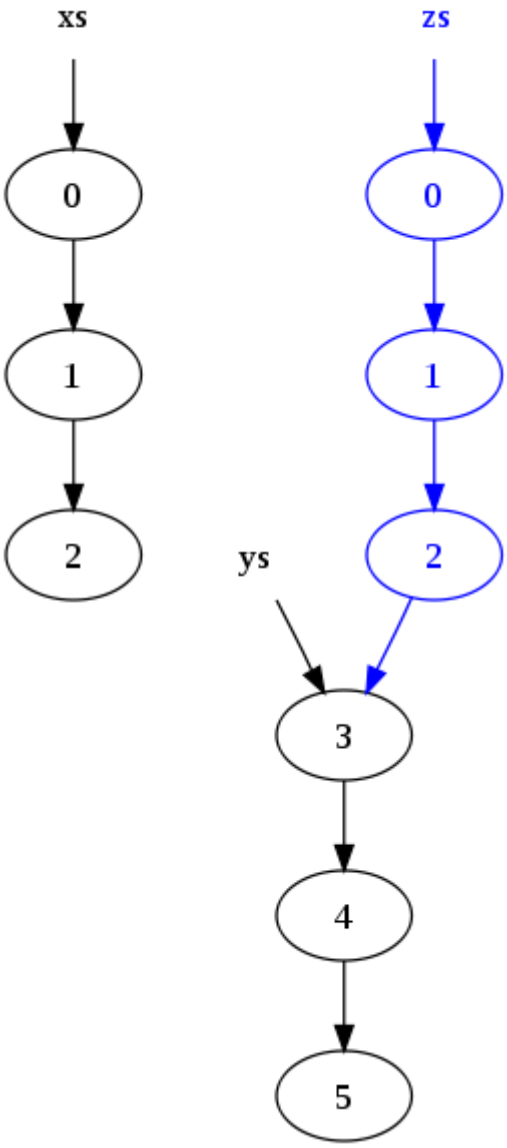


where a circle indicates a node in the list (the arrow out representing the second element of the node which is a pointer to another node).

Now concatenating the two lists:

```
zs = xs ++ ys
```

results in the following memory structure:



Notice that the nodes in list `xs` have been copied, but the nodes in `ys` are shared. As a result, the original lists (`xs` and `ys`) persist and have not been modified.

The reason for the copy is that the last node in `xs` (the node containing the original value `2`) cannot be modified to point to the start of `ys`, because that would change the value of `xs`.

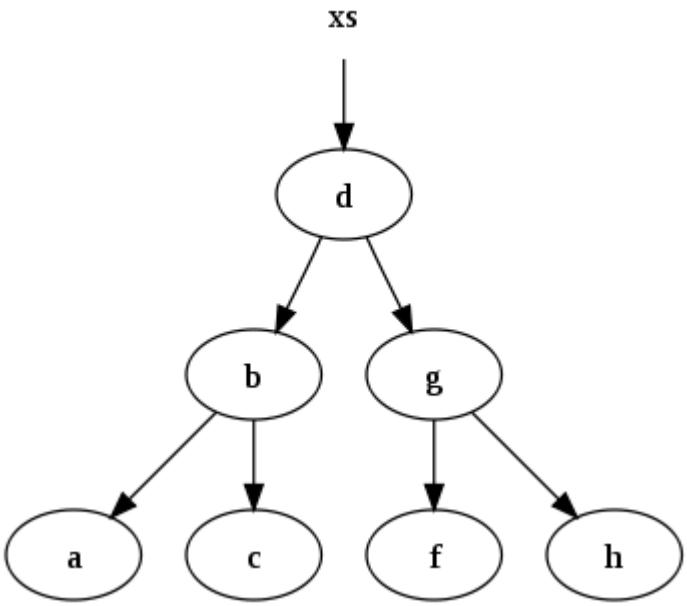
Trees

Consider a binary search tree,[9] where every node in the tree has the recursive invariant that all subnodes contained in the left subtree have a value that is less than or equal to the value stored in the node, and subnodes contained in the right subtree have a value that is greater than the value stored in the node.

For instance, the set of data

```
xs = [a, b, c, d, f, g, h]
```

might be represented by the following binary search tree:



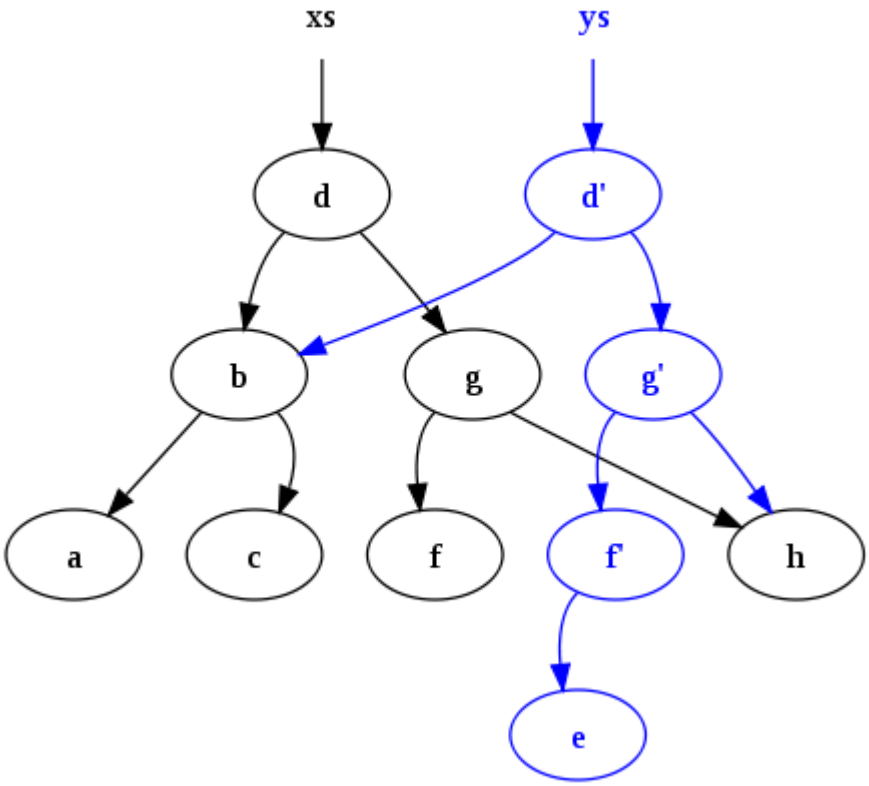
A function which inserts data into the binary tree and maintains the invariant is:

```
fun insert (x, E) = T (E, x, E)
  | insert (x, s as T (a, y, b)) =
      if x < y then T (insert (x, a), y, b)
      else if x > y then T (a, y, insert (x, b))
      else s
```

After executing

```
ys = insert ("e", xs)
```

The following configuration is produced:



Notice two points: first, the original tree (`xs`) persists. Second, many common nodes are shared between the old tree and the new tree. Such persistence and sharing is difficult to manage without some form of garbage collection (GC) to automatically free up nodes which have no live references, and this is why GC is a feature commonly found in functional programming languages.

Persistent hash array mapped trie

A persistent hash array mapped trie is a specialized variant of a hash array mapped trie that will preserve previous versions of itself on any updates. It is often used to implement a general purpose persistent map data structure.[10]

Hash array mapped tries were originally described in a 2001 paper by Phil Bagwell entitled "Ideal Hash Trees". This paper presented a mutable Hash table where "Insert, search and delete times are small and constant, independent of key set size, operations are O(1). Small worst-case times for insert, search and removal operations can be guaranteed and misses cost less than successful searches".[11] This data structure was then modified by Rich Hickey to be fully persistent for use in the Clojure programming language.[12]

Conceptually, hash array mapped tries work similar to any generic tree in that they store nodes hierarchically and retrieve them by following a path down to a particular element. The key difference is that Hash Array Mapped Tries first use a hash function to transform their lookup key into a (usually 32 or 64 bit) integer. The path down the tree is then determined by using slices of the binary representation of that integer to index into a sparse array at each level of the tree. The leaf nodes of the tree behave similar to the buckets used to construct hash tables and may or may not contain multiple candidates depending on hash collisions.[10]

Most implementations of persistent hash array mapped tries use a branching factor of 32 in their implementation. This means that in practice while insertions, deletions, and lookups into a persistent hash array mapped trie have a computational complexity of $O(\log n)$, for most applications they are effectively

constant time, as it would require an extremely large number of entries to make any operation take more than a dozen steps.[13]

Usage in programming languages

Haskell

Haskell is a pure functional language and therefore does not allow for mutation. Therefore, all data structures in the language are persistent, as it is impossible to not preserve the previous state of a data structure with functional semantics.[14] This is because any change to a data structure that would render previous versions of a data structure invalid would violate referential transparency.

In its standard library Haskell has efficient persistent implementations for linked lists,[15] Maps (implemented as size balanced trees),[16] and Sets[17] among others.[18]

Clojure

Like many programming languages in the Lisp family, Clojure contains an implementation of a linked list, but unlike other dialects its implementation of a Linked List has enforced persistence instead of being persistent by convention.[19] Clojure also has efficient implementations of persistent vectors, maps, and sets based on persistent hash array mapped tries. These data structures implement the mandatory read-only parts of the Java collections framework.[20]

The designers of the Clojure language advocate the use of persistent data structures over mutable data structures because they have value semantics which gives the benefit of making them freely shareable between threads with cheap aliases, easy to fabricate, and language independent.[21]

These data structures form the basis of Clojure's support for parallel computing since they allow for easy retries of operations to sidestep data races and atomic compare and swap semantics.[22]

Elm

The Elm programming language is purely functional like Haskell, which makes all of its data structures persistent by necessity. It contains persistent implementations of linked lists as well as persistent arrays, dictionaries, and sets.[23]

Elm uses a custom virtual DOM implementation that takes advantage of the persistent nature of Elm data. As of 2016 it was reported by the developers of Elm that this virtual DOM allows the Elm language to render HTML faster than the popular JavaScript frameworks React, Ember, and Angular.[24]

Java

The Java programming language is not particularly functional. Despite this, the core JDK package java.util.concurrent includes CopyOnWriteArrayList and CopyOnWriteArraySet which are persistent structures, implemented using copy-on-write techniques. The usual concurrent map implementation in Java, ConcurrentHashMap, is not persistent, however. Fully persistent collections are available in third-party libraries, or other JVM languages.

JavaScript

The popular JavaScript frontend framework React is frequently used along with a state management system that implements the Flux architecture,[25][26] a popular implementation of which is the JavaScript library Redux. The Redux library is inspired by the state management pattern used in the Elm programming language, meaning that it mandates that users treat all data as persistent.[27] As a result, the Redux project recommends that in certain cases users make use of libraries for enforced and efficient persistent data structures. This reportedly allows for greater performance than when comparing or making copies of regular JavaScript objects.[28]

One such library of persistent data structures Immutable.js is based on the data structures made available and popularized by Clojure and Scala.[29] It is mentioned by the documentation of Redux as being one of the possible libraries that can provide enforced immutability.[28] Mori.js brings data structures similar to those in Clojure to JavaScript.[30] Immer.js brings an interesting approach where one "creates the next immutable state by mutating the current one". [31] Immer.js uses native JavaScript objects and not efficient persistent data structures and it might cause performance issues when data size is big.

Prolog

Prolog terms are naturally immutable and therefore data structures are typically persistent data structures. Their performance depends on sharing and garbage collection offered by the Prolog system.[32] Extensions to non-ground Prolog terms are not always feasible because of search space explosion. Delayed goals might mitigate the problem.

Some Prolog systems nevertheless do provide destructive operations like setarg/3, which might come in different flavors, with/without copying and with/without backtracking of the state change. There are cases where setarg/3 is used to the good of providing a new declarative layer, like a constraint solver.[33]

Scala

The Scala programming language promotes the use of persistent data structures for implementing programs using "Object-Functional Style".[34] Scala contains implementations of many Persistent data structures including Linked Lists, Red–black trees, as well as persistent hash array mapped tries as introduced in Clojure.[35]

Garbage collection

Because persistent data structures are often implemented in such a way that successive versions of a data structure share underlying memory[36] ergonomic use of such data structures generally requires some form of automatic garbage collection system such as reference counting or mark and sweep.[37] In some platforms where persistent data structures are used it is an option to not use garbage collection which, while doing so can lead to memory leaks, can in some cases have a positive impact on the overall performance of an application.[38]

See also

- Copy-on-write
- Navigational database
- Persistent data

- Retroactive data structures
- Purely functional data structure

References

1. Driscoll JR, Sarnak N, Sleator DD, Tarjan RE (1986). "Making data structures persistent". *Proceedings of the eighteenth annual ACM symposium on Theory of computing - STOC '86*. *Proceeding STOC '86. Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. pp. 109–121. CiteSeerX 10.1.1.133.4630 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.133.4630). doi:10.1145/12130.12142 (https://doi.org/10.1145%2F12130.12142). ISBN 978-0-89791-193-1. S2CID 364871 (https://api.semanticscholar.org/CorpusID:364871).
2. Kaplan, Haim (2001). "Persistent data structures" (http://www.math.tau.ac.il/~haimk/papers/persistent-survey.ps). *Handbook on Data Structures and Applications*.
3. Conchon, Sylvain; Filliâtre, Jean-Christophe (2008), "Semi-persistent Data Structures", *Programming Languages and Systems*, Lecture Notes in Computer Science, **4960**, Springer Berlin Heidelberg, pp. 322–336, doi:10.1007/978-3-540-78739-6_25 (https://doi.org/10.1007%2F978-3-540-78739-6_25), ISBN 9783540787389
4. Tiark, Bagwell, Philip Rompf (2011). *RRB-Trees: Efficient Immutable Vectors*. OCLC 820379112 (https://www.worldcat.org/oclc/820379112).
5. Brodal, Gerth Stølting; Makris, Christos; Tsichlas, Kostas (2006), "Purely Functional Worst Case Constant Time Catenable Sorted Lists", *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 172–183, CiteSeerX 10.1.1.70.1493 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.1493), doi:10.1007/11841036_18 (https://doi.org/10.1007%2F11841036_18), ISBN 9783540388753
6. Neil Sarnak; Robert E. Tarjan (1986). "Planar Point Location Using Persistent Search Trees" (https://web.archive.org/web/20151010204956/http://www.link.cs.cmu.edu/15859-f07/papers/point-location.pdf) (PDF). *Communications of the ACM*. **29** (7): 669–679. doi:10.1145/6138.6151 (https://doi.org/10.1145%2F6138.6151). S2CID 8745316 (https://api.semanticscholar.org/CorpusID:8745316). Archived from the original (http://www.link.cs.cmu.edu/15859-f07/papers/point-location.pdf) (PDF) on 2015-10-10. Retrieved 2011-04-06.
7. Chris Okasaki. "Purely Functional Data Structures (thesis)" (https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf) (PDF).
8. Liljenzin, Olle (2013). "Confluently Persistent Sets and Maps". arXiv:1301.3388 (https://arxiv.org/abs/1301.3388). Bibcode:2013arXiv1301.3388L (https://ui.adsabs.harvard.edu/abs/2013arXiv1301.3388L).
9. *This example is taken from Okasaki. See the bibliography.*
10. BoostCon (2017-06-13), *C++Now 2017: Phil Nash "The Holy Grail!? A Persistent Hash-Array-Mapped Trie for C++"* (https://www.youtube.com/watch?v=WT9kmIE3Uis), retrieved 2018-10-22
11. Phil, Bagwell (2001). "Ideal Hash Trees" (https://infoscience.epfl.ch/record/64398).
12. "Are We There Yet?" (https://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey). *InfoQ*. Retrieved 2018-10-22.
13. Steindorfer, Michael J.; Vinju, Jurgen J. (2015-10-23). "Optimizing hash-array mapped tries for fast and lean immutable JVM collections" (https://ir.cwi.nl/pub/24029). *ACM SIGPLAN Notices*. **50** (10): 783–800. doi:10.1145/2814270.2814312 (https://doi.org/10.1145%2F2814270.2814312). ISSN 0362-1340 (https://www.worldcat.org/issn/0362-1340). S2CID 10317844 (https://api.semanticscholar.org/CorpusID:10317844).
14. "Haskell Language" (https://www.haskell.org/). *www.haskell.org*. Retrieved 2018-10-22.
15. "Data.List" (http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html). *hackage.haskell.org*. Retrieved 2018-10-23.

16. "Data.Map.Strict" (http://hackage.haskell.org/package/containers-0.6.0.1/docs/Data-Map-Strict.html). *hackage.haskell.org*. Retrieved 2018-10-23.
17. "Data.Set" (http://hackage.haskell.org/package/containers-0.6.0.1/docs/Data-Set.html). *hackage.haskell.org*. Retrieved 2018-10-23.
18. "Performance/Arrays - HaskellWiki" (https://wiki.haskell.org/Performance/Arrays). *wiki.haskell.org*. Retrieved 2018-10-23.
19. "Clojure - Differences with other Lisps" (https://clojure.org/reference/lisps). *clojure.org*. Retrieved 2018-10-23.
20. "Clojure - Data Structures" (https://clojure.org/reference/data_structures). *clojure.org*. Retrieved 2018-10-23.
21. "Keynote: The Value of Values" (https://www.infoq.com/presentations/Value-Values). *InfoQ*. Retrieved 2018-10-23.
22. "Clojure - Atoms" (https://clojure.org/reference/atoms). *clojure.org*. Retrieved 2018-11-30.
23. "core 1.0.0" (https://package.elm-lang.org/packages/elm/core/latest/). *package.elm-lang.org*. Retrieved 2018-10-23.
24. "blog/blazing-fast-html-round-two" (https://elm-lang.org/blog/blazing-fast-html-round-two). *elm-lang.org*. Retrieved 2018-10-23.
25. "Flux | Application Architecture for Building User Interfaces" (https://facebook.github.io/flux/). *facebook.github.io*. Retrieved 2018-10-23.
26. Mora, Osmel (2016-07-18). "How to handle state in React" (https://medium.com/react-ecosystem/how-to-handle-state-in-react-6f2d3cd73a0c). *React Ecosystem*. Retrieved 2018-10-23.
27. "Read Me - Redux" (https://redux.js.org/). *redux.js.org*. Retrieved 2018-10-23.
28. "Immutable Data - Redux" (https://redux.js.org/faq/immutabledata). *redux.js.org*. Retrieved 2018-10-23.
29. "Immutable.js" (https://web.archive.org/web/20150809185757/http://facebook.github.io/immutable-js/). *facebook.github.io*. Archived from the original (https://facebook.github.io/immutable-js/) on 2015-08-09. Retrieved 2018-10-23.
30. "Mori" (https://swannodette.github.io/mori/).
31. "Immer" (https://github.com/immerjs/immer). 26 October 2021.
32. Djamboulian, Ara M.; Boizumault, Patrice (1993), *The Implementation of Prolog - Patrice Boizumault* (https://press.princeton.edu/books/hardcover/9780691637709/the-implementation-of-prolog), ISBN 9780691637709
33. *The Use of Mercury for the Implementation of a Finite Domain Solver - Henk Vandecasteele, Bart Demoen, Joachim Van Der Auwera* (https://lirias.kuleuven.be/retrieve/440562), 1999
34. "The Essence of Object-Functional Programming and the Practical Potential of Scala - codecentric AG Blog" (https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/). *codecentric AG Blog*. 2015-08-31. Retrieved 2018-10-23.
35. ClojureTV (2013-01-07), *Extreme Cleverness: Functional Data Structures in Scala - Daniel Spiewak* (https://www.youtube.com/watch?v=pNhBQJN44YQ), retrieved 2018-10-23
36. "Vladimir Kostyukov - Posts / Slides" (https://kostyukov.net/posts/designing-a-pfds/). *kostyukov.net*. Retrieved 2018-11-30.
37. "http://wiki.c2.com/?ImmutableObjectsAndGarbageCollection" (http://wiki.c2.com/?ImmutableObjectsAndGarbageCollection). *wiki.c2.com*. Retrieved 2018-11-30. External link in `|title=` (help)
38. "The Last Frontier in Java Performance: Remove the Garbage Collector" (https://www.infoq.com/news/2017/03/java-epsilon-gc). *InfoQ*. Retrieved 2018-11-30.

Further reading

External links

- Lightweight Java implementation of Persistent Red-Black Trees (http://wiki.edinburghhacklab.com/PersistentRedBlackTreeSet)
- Efficient persistent structures in C# (https://persistent.codeplex.com/)