Contents

5.3 Confluence

Functions acting simultaneously on an Immutable Data Structure each create different versions of the structure. These versions may be combined, provided they are not the result of conflicting functions, to create a new version which is equivalent to a serial execution of the functions. A function that combines past versions of a data structure is called a meld function. The existence of a meld function endows an Immutable Data Structure with the property of confluent persistence.

The validate function can ensure that two functions acting on an Immutable Data Structure do not contain conflicting operations and that combining the two versions produced in isolation will result in a single version which is equivalent to a serial execution of the functions. The problem is how to combine these versions into a single version?

The main contribution of this section is the description of a meld function that combines versions of an Immutable Data Structure produced in isolation. This section focuses on techniques for making Immutable Data Structures confluently persistent.

5.3.1 Simultaneous modifications

To make the functions of our Immutable Data Structure confluently persistent we need a meld function that can combine two non-conflicting versions of the Immutable Data Structure.

For example, consider two functions acting simultaneously on a deque. One function inserts an element onto the back of the queue and a second function, executing on another processor, simultaneously removes an element from the front of the queue. The functions do not conflict but they do result in two different versions of the queue which must be melded to produce a new version of the queue which is equivalent to a version produced by a serial execution of the functions.

5.3.2 Meld Function

The meld function takes two versions of the Canonical Binary Tree and creates a new version by full copying the nodes corresponding to variables in the cap. When used in conjunction with a validate function that rejects conflicting operations

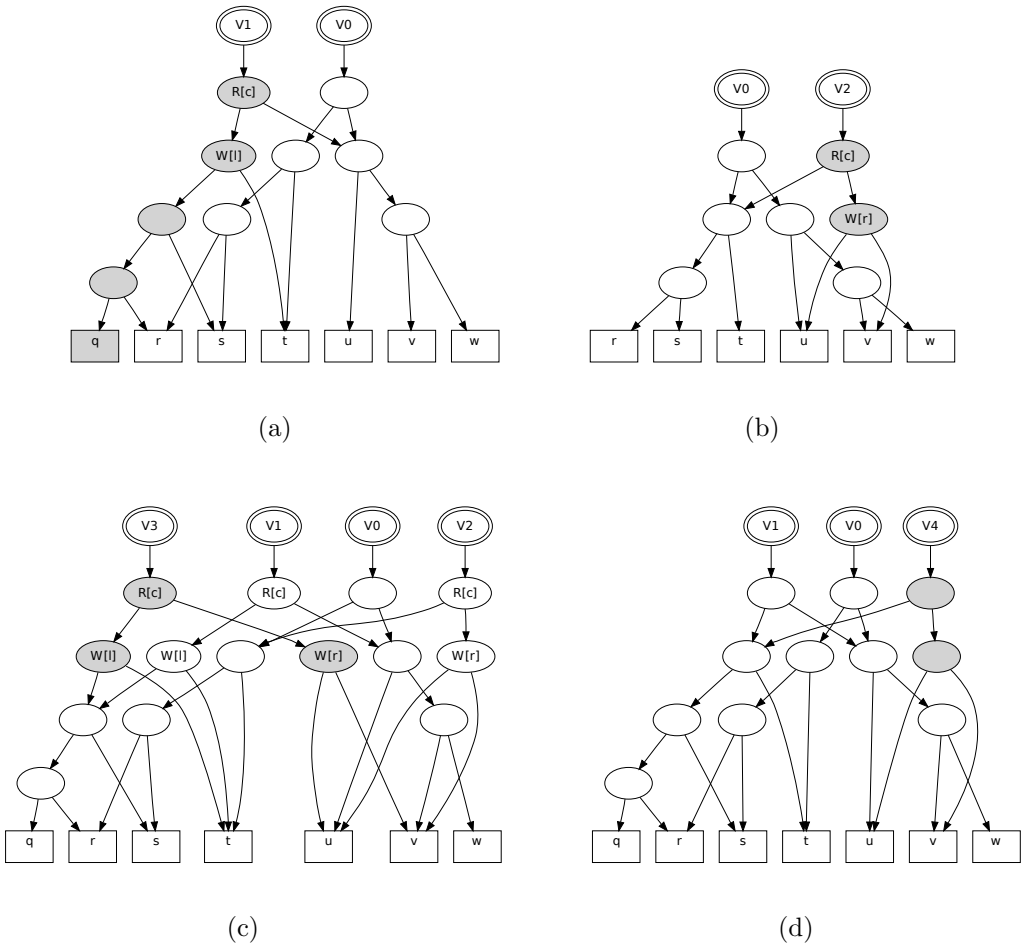

Figure 5.1: **Making a deque confluently persistent** by using a meld function that combines versions created in isolation. The cap of a deque contains three variables **l**, **r** and **c**.
(a) The function $Push\_front(q)$ acts on version V0 which contains values $\{r, s, t, u, v, w\}$. It creates version V1 which contains the values $\{q, r, s, t, u, v, w\}$. The operations $\{\mathbf{R}[\mathbf{c}], \mathbf{W}[\mathbf{l}]\}$ are recorded in the path.
(b) The function $Pop\_back()$ acts on version V0 to create version V2 which contains $\{r, s, t, u, v\}$. The operations $\{\mathbf{R}[\mathbf{c}], \mathbf{W}[\mathbf{r}]\}$ are recorded in the path.
(c) Versions V1 and V2 meld to produce a new version V3 which contains $\{q, r, s, t, u, v\}$. The meld selectively copies the nodes from versions V1 and V2.
(d) A serial execution of these functions would have created version V4 which contains $\{q, r, s, t, u, v\}$ and is equivalent to V3.

the meld function makes an Immutable Data Structure confluently persistent.

The meld function takes references to versions of arbitrary complexity as its parameters and returns a reference to a new version. It is specialised by a parameter representing the topology of the cap. The meld function traverses the nodes in the cap of both versions and compares the operations acting on nodes corresponding to the same variable. The meld function is a full copy operation that selectively copies nodes from the two versions using the operations and time stamps recorded in the nodes to determine which version to copy. For each variable in the cap a new node is created to represent it. The function returns a reference to a root node which represents a new version.

Figure 5.1 illustrates an example of how a deque can be made confluently persistent by a meld function that combines two versions simultaneously created in isolation.

The two versions in this example could have been combined by creating a new root node. However, in the general case it is necessary to copy all of the nodes which correspond to variables in the cap to ensure that the correct time stamps are recorded in the nodes and that the relationship between operations and variables is maintained.

The two versions in this example could have been combined without using time stamps because each version represents the action of a single function. However, in the general case it is necessary to consider the time stamps associated with each node while performing the full copy.

The deque is a particularly simple example, however all ADTs implemented by the Canonical Binary Tree can be made confluently persistent using the technique. The meld function is implemented by the Canonical Binary Tree, the topology of the cap is supplied as a parameter but the operation of the meld function is ADT agnostic.

5.3.3 Previous work

Driscoll defines confluence in a seminal paper on persistent data structures [DSST86].

Versions of an Immutable Data Structure created by arbitrary transformations cannot always be combined because the functions that created them may conflict. Fiat considers that the problem of making a data structure confluently persistent is intractable in the general case [FK03]. When conflicting functions are eliminated the problem of implementing a meld function becomes tractable,

but even functions that do not conflict can transform the topology of a data structure in ways that are difficult to reconcile.

Version control systems for documents are well known applications of confluent persistence. Pilato describes a system called Subversion which enables multiple authors to modify a document concurrently [PCSF08]. Subversion provides a validate function that highlights any conflicting modifications to a document and a meld operation which combines multiple versions

Bibliography

[DSST86] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM.

[FK03] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *J. Algorithms*, 48(1):16–58, 2003.

[PCSF08] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2 edition, September 2008.