Consistency in Distributed Databases

A Group-like Algebra and its Applications

A.W. Roscoe

Oxford University Computing Laboratory

This paper appeared as Oxford University Computing Laboaratory Technical Monograph PRG-87 (1990).

Abstract

We introduce, and prove correct, two novel algorithms for preserving a form of consistency in distributed databases arranged as rings. The first uses as its model databases with a fixed number of fields with updates which assign known constant values to one of these 'slots'. The proof of this relies on a moderately complex combinatorial argument. The second algorithm, which can be viewed as generalising the first, takes a wider view and simply assumes that the set of updates have an operation analogous to the conjugation of group theory: given any u, v we can find u^v such that $u; v = v; u^v$, which satisfies some natural algebraic properties. Its proof relies on an algebraic argument based on partial orders, which may well have applications outside databases, for example in the field of 'true concurrency'. We indicate how the algorithm can be generalised to a number of other network topologies, and give guidelines for further generalisations. If combined with timestamping, the algorithms provide highly concurrent methods of ensuring that the sequence of updates executed at all nodes corresponds to the order implied by these timestamps.

1 Introduction

Distributed databases, where multiple copies of some data are kept in different locations, occur in a wide range of applications, varying from multiple copies of a cache memory in shared-variable parallel computers, through networks of workstations sharing some common information, to widely distributed applications such as automated teller machines. The multiple copies may be kept for speed of access, for security (i.e., process P does not lose vital data if process Q goes down), or a combination of the two.

Our conceptual model will be of a number of processes with separate memory, all seeking to hold copies of the same database. The database is changed via updates, which are circulated around the network and executed by the processes (presumably as they arrive at each one).

An obvious problem arises from the need to keep the various copies of a piece of data consistent: if several processors decide at more-or-less the same time to update it, how can we ensure that all processors, at least ultimately, agree on its value? Given the principles (i) that any process can update any location, and (ii) that each process should be allowed to execute, locally, its own updates immediately (the denial of which would lead to some interesting programming problems), it seems inevitable that there will be times when the various copies will not agree. We will seek a weaker form of consistency, namely that whenever there are no updates circulating which affect some portion of the database, then all copies of this portion are the same. Thus if the network is *quiescent* (i.e., no updates at all are queued or circulating) then all copies of the database are equal. A secondary but no less necessary requirement is that, if no new updates are inserted after any time, then the network will become quiescent in a reasonable period of time.

The author's work on this topic began in 1983 when he encountered a commercial solution to this problem. A computer manufacturer was introducing a system to manage networked workstations, part of which required the consistency of copies of a database stored in each node. In broad outline, their solution was to arrange the workstations in a ring. (This, historically, is the reason why most of the author's work has been based on rings. We will see in the Section 6 that his choice of a ring topology is by no means essential.) Their ring contains exactly one 'token', which either carries a single update round all the nodes or is empty. Thus a given node can only insert an update into the ring (for execution by the other processors) when it can acquire the token, empty. Each node is allowed to execute its own updates immediately they arise, and these are then queued for insertion into the ring. In order to avoid the inconsistencies that potentially arise in the manner stated above, it was necessary to restrict the model of updates to the assignment of a constant value to a single slot (i.e., store location). When an update arrives from the ring which clashes with one(s) queued locally (i.e., is an assignment to the same slot), these are removed from the queue and the newly arrived one is executed.

This system transparently achieves the desired correctness, but the fact that only one update can circulate at a time is potentially very limiting. Imagine scaling this system: if there are N processes, then the total number of updates generated is likely to grow proportionately with N, but since the time for an update to circulate will also grow with N, the throughput will be proportional to $1/N$.

The algorithm presented in Section 2 was devised at that time, but the correctness proof was not completed, and nor was the algorithm written up. The author's interest was revived recently when looking for natural problems for combining Z (a specification language for state-based systems such as databases) and CSP (a language and theory for reasoning about distributed systems). The application of these methods to the problem may be reported later. He then completed the proof given in Section 2, and produced the generalisation presented in Section 4.

The generalised algorithm and its proof are based on an algebra with an associative sequential composition operator $u; v$ and a 'conjugation' operator u^v such that $v; u^v = v; u$. By-products of this work are two interesting algebraic theory, which we term 'conjugate algebras' and 'box algebras'. The first of these is a natural generalisation of groups, and the second (though discovered independently) bears a close resemblence to Stark's [S] 'algebra of residuals'. The proof of the generalised algorithm depends on the (new) result that it is possible to capture the structure of any finite partial order naturally within this second algebra. It seems possible that this work might find further applications in the field of 'true concurrency', where process histories come with a partial order describing the causal dependence between events – interestingly it is precisely this order which we have to characterise algebraically to prove our algorithm. This was, in any case, the field which motivated Stark's work.

Several other algorithms for handling the database problem we address are based on the idea of *timestamping*: each update is marked with time when it was generated and these times are used by nodes to piece together the correct order to execute updates. Timestamping is not essential in our framework, but we show at the end of Section 4 how a priority mechanism based on timestamps can be used to achieve attractive results.

In Section 5 we investigate some more properties of conjugate algebras and give some non-trivial examples.

In Section 6 we address the question of how our algorithms might be adapted to non-ring topologies. We show that they readily extend to networks arranged either as trees or as multiple rings interconnected as a tree. There is some discussion of how the algebraic constructions used in the proof of the earlier algorithms might allow us to derive algorithms for more general networks.

Finally, in Section 7, we point to a few topics that might be worthy of further work.

2 Concurrent simple updates

The algorithm described in the introduction can be modified in a fairly straightforward way to allow an arbitrary number of updates to circulate concurrently without affecting correctness. As we will find, what it not straightforward is verifying that this is in fact the case.

The modified algorithm – which will be referred to as *Algorithm 1* – replaces the token ring with a more-or-less arbitrary no-overtaking ring, which can allow as many updates to circulate as it pleases, but must never let one update overtake another. The nodes on the ring are assigned arbitrary but unique priorities. The effect of giving a node higher priority is that its updates will be slightly more likely to be executed across the network: there may be some external reason for ordering the processes, but if not they must be prioritised anyway. The priority of node N_i will be denoted p_i, and $p_i > p_j$ will mean that N_i has higher priority than N_j.

As in the case of the algorithm described in the introduction, we will suppose that each node N_i has a queue Q_i of updates, executed locally but not yet entered into the ring. If desired, these can now be dispensed with, since all that would be required would be for N_i to 'generate' an extra place actually on the ring within itself: whether the Q_i's are used will depend on hardware configurations and efficiency analysis. We have retained the possibility of them in our algorithms to demonstrate that, if properly managed, they do not damage correctness.

Each node N_i will also have a queue E_i, which will contain a list of all updates which it has inserted into the ring and is expecting back. This is not dispensable.

As in the algorithm described in the introduction, our model of updates will be the assignment of constant values to single slots. Such an update u can thus be written $x := c$. Notice that, given updates $u \equiv x := c$ and $v \equiv x' := c'$, we either have $u; v = v; u$ (when x and x' are distinct), or have $u; v = v$.

We describe the algorithm in terms of the actions of an individual node N_i. (The actions within a node are assumed to be atomic, in that, once one is started, it is completed without any others overlapping it.)

- If N_i generates an update u, then u is executed locally and u is inserted at the tail of Q_i.
- If Q_i is nonempty and there is a space available on the ring, then the head of Q_i is removed and inserted into the ring, and into the tail of E_i.
- If an update u returns which was originated by N_i, then it is removed from the ring, and from the head of E_i.[1]

[1]It is by no means obvious, given the details of this algorithm, that u will still be in E_i and, if it is, whether it will be at the head. Until we have established that the above statement actually makes sense we will say that u is removed from the ring only if it is still expected back and, if so, is removed from E as well. Notice that this leaves the formal possibility that an update might pass its origin and circle the ring indefinitely. However, it will transpire from our later analysis of this algorithm that u will indeed be at the head of E_i, so that the formulation above is correct.

- If an update u arrives that originated at some other N_j and such that there is no clashing update (one to the same slot) in E_i, then u is executed locally and any clashing updates in Q_i are deleted. u is passed on round the ring.
- If an update u arrives that originated at N_j with $p_j < p_i$, and E_i contains one(s) that clash with it, then u is neither executed locally, nor is it passed round the ring. We say that it is *stopped*.
- If an update u arrives that originated at N_j with $p_j > p_i$, and E_i contains ones that clash with it, then these are removed from E_i (they are *cancelled*), u is executed locally and any clashing updates in Q_i are removed. u is passed on round the ring.

Clearly, we are using the regime of stopping updates and cancelling their expected-back versions to maintain consistency in the presence of multiple updates to the same slot.

If we want to implement the above algorithm (and the second algorithm described later) it is clear that the update transported round the ring needs to carry with it not only the operator which updates the state, but also a unique identifier from which a node can identify the priority of the update, and tell whether it was one which the node itself generated. (The point is that there may be many functionally identical updates circulating at any one time, which for our purposes need to be distinguished.) In describing our algorithms we will describe only what happens to the functional part of an update, but the reader should bear in mind that they will always carry this unique identifier with them as well.

In order to get reasonable behaviour of this system we have to make some stipulations about the behaviour of the ring itself.

- It must have bounded capacity. This is to ensure that no update can make an infinite amount of progress without getting all the way round.
- It does not indefinitely stop any update within it from making progress. (This could conceivably happen either because the system can deadlock, or because it can favour one section of the ring in preference to others.)
- Provided enough items are removed from it, the ring will not indefinitely delay accepting any update which a node wishes to insert. This is easily achieved by, for example, reserving at least one space in the ring specifically for updates generated by each node.

There are numerous ways of achieving these goals, for example a synchronously clocked ring where all places (empty or full) move round in lockstep, or a small modification of the ring described in [R].

The chief aim of the rest of this section will be to prove the following result.

Theorem 2.1 Suppose initially all copies of the database are identical. Under Algorithm 1, whenever there is no current update (one in the ring or any Q_i) to a given slot, all nodes agree on the value in that slot. Furthermore, if after a given time no further updates are generated for a slot, then by some later time there will be no current update for it. ∎

Our assumptions about the ring are not necessary to prove the first part of this result, but clearly they will be vital in establishing the second.

From here on it will be useful to make three simplifying assumptions. First, that there is actually only one slot (so that all updates clash), second, that there are no Q_i's, and finally, that no N_i emits more than one update in its history. We will justify each of these in turn. In understanding these and later arguments it might be helpful to bear in mind that we are never, in this paper, interested in what *caused* a particular update to be generated. Thus, if we alter the sequence of updates executed at a given node, we can ignore the possibility that this might change the subsequent updates it generates. All we interested in doing is proving our consistency results in the presence of any possible sequence of updates arising from each node.

The assumptions we have made about the ring mechanism mean that updates to one slot cannot prevent those to another getting round the ring. Any sequence of updates to a given slot that would be possible in the presence of those to other slots is equally possible without them, and *vice versa*. For the algorithm we have set up treats updates to different slots independently: they never stop, cancel or otherwise affect each other. Thus, to prove the above theorem, it will be enough to prove that it holds if updates are restricted to a single slot. Notice that, under this assumption, the network is quiescent exactly when there are no current updates for the single slot.

The effect of any sequence of updates in the algorithm with the Q_i's is, for our purposes, equivalent to one without them. This sequence is the one where each node executes its own updates only when it they are inserted into the ring, and these events only occur for updates which, in the original sequence, were not removed from Q_i before insertion. Notice that this does not affect the contents of the E_i or the updates that circulate in the ring: both of these are at every time exactly as they were in the original. This manipulation affects the short-term behaviour of N_i's copy of the database, since it is not as up-to-date and may not see some locally generated values. But it does not affect the long-term behaviour, in the sense that the final value of each variable will be the same as in the original. For if the last update u to be executed in the old sequence, at a given node N, was generated outside N, then this will be executed in the new sequence and there will be no later local updates executed there since any that were queued up at the time when u arrives are deleted. And if it was generated at N then no later update arrives to delete it and stop it being executed in the new sequence.

Suppose we have proved Theorem 2.1 for systems where each node is only allowed to emit one update. Suppose further that we have proved (as we will) (i) that, in this case, any update is stopped if and only if it is cancelled (i.e., the copy expected back by its originator is) and (ii) that all stopping and cancelling takes place before an update would otherwise have returned to its origin. Notice that the things we are required to prove in the theorem (for our single-slot system) are all on the assumption that the number of updates inserted is finite. So suppose that, on a given occasion, the number of updates inserted by each node is bounded above by r. We will show that this behaviour of the system is modelled by one of the system where (i) each N_i is replaced by r adjacent nodes $N_{i,j}$ $(1 \leq j \leq r)$, limited to one update each, and (ii) the priorities of the $N_{i,j}$ are arranged so that, if $i \neq i'$, then $p_{i,j} < p_{i',j'}$ if, and only if, $p_i < p_{i'}$ (in other words, the priorities of the $N_{i,j}$ are 'small perturbations' of those of the N_i).

The $N_{i,j}$ are arranged on the ring so that j increases as we proceed round the ring in the same direction as updates are transmitted. $N_{i,1}$ is given the job of transmitting the first update that N_i puts into the ring, $N_{i,2}$ the second, and so on for as long as necessary. Since no node in the original system is ever dealing with more than one update at a time, we can model its behaviour by one in the revised system where, at any one time, the portion of the ring from $N_{i,1}$ to $N_{i,r}$ (inclusive) never contains more than one update. Thus we will ensure that the previous update to be dealt with has left $N_{i,r}$, has been stopped, or has been removed by its originator, before we allow another update to be inserted by one of this sequence of nodes or to enter $N_{i,1}$

from the ring[2]. By doing this, it is clear that we can regard the handling of an update by one of these sequences as atomic, and directly model the behaviour of the original system. Because of this exclusion principle and the order in which they are generated, no update generated by one member of a sequence $N_{i,1}, \ldots N_{i,r}$ ever stops or cancels another. (The two facts (i) and (ii) about stopping and cancelling stated above are necessary to establish this.)

If we now regard the sequence of nodes $N_{i,1}$ to $N_{i,r}$ as being, in some sense, a unit, whose list E_i of expected-back updates is those expected by these in order (lowest index at the head), then the behaviour of the original system can be modelled directly in the new one. The new system will stop or cancel updates exactly as the old one did, and furthermore node $N_{i,r}$ (i.e., the last in the sequence) sees exactly the same sequence of updates as was seen by N_i in the original one. For each update generated by one of the $N_{i,j}$ is seen there, by construction, and any update generated by another node which enters the sequence will have been stopped by the time it, or its 'ghost[3]', reaches $N_{i,r}$ if and only if it would have been stopped at N_i. The result for the original system now follows directly from the one we have assumed for the new one.

From now on we will make the three simplifying assumptions. As a very simple illustration of how the algorithm works, consider the case of a ring of three nodes, all of which emit their update at the same time (or, at least, before any other updates reach them). To be specific, assume that, in the direction of the ring, their priorities are respectively 1,2,3 (3 highest) and that the updates they emit are u_1, u_2 and u_3 respectively. In this experiment, u_1 is stopped by N_2, and u_2 is stopped by N_3. u_3 cancels both u_1 and u_2, and is the last update seen by all three nodes. Thus they all agree on the value assigned by u_3. One point to note here is that u_1 is stopped by a different node than the update that cancels it: in larger rings the dependencies between the updates that stop and cancel a given update can become long and diffuse. Notice here that, in a symmetrical situation, the fact that u_3 has the highest priority has led to it 'winning'. If, however, we break the symmetry by assuming that u_3 has passed N_1 before u_1 and u_2 are simultaneously emitted, we see that u_3 cancels u_2, and N_3 stops u_2; but u_1 is neither stopped nor cancelled and is, in fact, the last update seen by all three nodes – and so 'wins'.

The reader might like to experiment with the algorithm on some slightly larger rings, under various assumptions about the order in which updates are transmitted. The subtlety of the way in which the algorithm works should become clear, as should the important fact that the overall behaviour of the system is determined purely by the moments at which updates are inserted (or, more precisely, the arrangement of other updates between nodes at the moment when each is put into the ring). The no-overtaking property of the ring means that this completely determines the sequence of updates that arrive at each individual node.

The proof of correctness of the algorithm is broken into three lemmas, the chief of which (in terms of difficulty, at least) is the first. The statements of the lemmas are as follows; their proofs follow.

Lemma 2.2 An update is cancelled if and only if it is stopped, and all stopping and cancelling takes place before an update would have returned to its origin (i.e., before its ghost does return).

Lemma 2.3 If no further updates are inserted after some time, then the network will eventually

[2] We are not suggesting that the new ring would always enforce these strange restrictions, only that each behaviour of the original ring can be modelled by one of this ring where the restrictions happen to be obeyed.

[3] We can think of a stopped update as leaving behind a *ghost* which travels round the ring until it reaches home, or perhaps even longer. This makes sense in a no-overtaking ring, since by stating that something happens before the ghost of u arrives we know that it happens before any update that follows u arrives. It seems to be easier to state conditions like the above in terms of these imaginary ghosts rather than to formulate them accurately in other ways

reach a quiescent state where (a) no node expects an update back, and (b) the last event to occur was an update returning to its origin.

LEMMA 2.4 All nodes, in this quiescent state, agree on the value carried by the final update referred to in (b) above.

PROOF OF LEMMA 2.2 We prove this by induction on the number of nodes with higher priority than (the one emitting) a given update. The result is trivial when there are none of these, for the update cannot then be stopped or cancelled. So suppose it holds of all updates generated by higher-priority nodes than N_0 which emits update u_0. For our result it will be enough to show that (i) if u_0 is stopped before its return to N_0 then it is cancelled before its ghost returns and (ii) if u_0 is cancelled at N_0 before u_0 or its ghost returns, then u_0 is in fact stopped before its return. These are sufficient because if u_0 is stopped or cancelled then one of these must occur before it returns (or else it returns to N_0 which is expecting it, and it is removed); the above results then mean that they both do.

Suppose, for contradiction to the first part, that u_0 is stopped at N_1 before it returns to N_0, but is not cancelled by the required time. Since N_1 has higher priority than N_0, we know by induction that u_1 (the update generated by N_1) is no longer expected back at N_1 after it or its ghost returns. It follows that u_1 and u_0 must be in the relation $u_0 \| u_1$, which we define to mean that neither (or their ghost, if previously stopped) has passed the other's origin before the latter one was emitted. (For otherwise N_1 could not have stopped u_0.) Since N_1 has higher priority than N_0, if u_1 were to reach N_0 it would cancel u_0 and satisfy the requirement of the Lemma. Thus we may assume that u_1 is in its turn stopped by N_2 (of yet higher priority) such that $u_1 \| u_2$. (The fact that, when an update v of higher priority than u_0 is stopped at a node expecting w back, then $w \| v$, is a simple consequence of our inductive hypothesis. We will take the fact as read in the rest of this construction.) u_2 may (i) reach N_0 or (ii) be stopped before it gets there by a node N_3 which is expecting an update u_3 back. The same two possibilities apply to u_3 in the second case, and it is clear that we can go on constructing u_n and N_n of increasing priority closer and closer to N_0 as long as option (ii) applies. Thus, eventually, option (i) applies for some u_k. It is easy to see that this u_k arrives at N_0 before u_0's ghost, so since we know (because we are assuming u_0 is not cancelled on time) that u_k does not cancel u_0, it must actually reach N_0 before u_0 is emitted, or in other words it is ahead of u_0.

In fact, we will demonstrate that this situation is impossible, which will complete the proof of this half of the main induction.

Let u_k be as above. It should not be hard to see that, for $i < k-1$, if u_k is ahead of u_{i+1} then it is also ahead of u_i. In fact, u_k cannot be ahead of u_{k-1}, since we would then have that (i) N_k stops u_{k-1} and (ii), by induction, that u_k either returns to N_k (removing its expectation) or is cancelled before its ghost would have returned: either of these lead to a contradiction to u_k being ahead of u_{k-1}. To be specific let $0 \leq s < k-1$ be maximal so that u_k is ahead of u_s.

Suppose in general we have, so far, created a sequence (inevitably in ascending priority) of updates $u_0, \ldots, u_r, u_{r+1}, \ldots u_{k-1}, u_k$ with $r \geq 0$, and $s \in \{r, \ldots, k-2\}$ such that:

- In moving round the ring once, N_r, N_{r+1}, …, N_k appear in sequence.
- N_{i+1} stops u_i for all i.
- u_k is ahead of $u_r, \ldots, u_s$, but not of $u_{s+1}, \ldots u_{k-1}$.

(This is exactly what we have set up above with $r = 0$.)

Then we know u_k is stopped before it can reach (and cancel before u_s gets there) N_{s+1}, by N_{k+1}, say. The update u_{k+1} may or may not be ahead of u_{s+1}. If it is, define $k' = k+1$. If not, we know that u_{k+1} must be stopped before it reaches N_{s+1}, so we may repeat the construction. We simply go on iterating until (as must happen eventually) we get $u_{k'}$ which is ahead of u_{s+1}. In general (because of our assumptions and construction) if $u_{k'}$ is ahead of u_j with $s < i < j < k'$ then it is ahead of u_i. But, by exactly the same inductive argument as applied above, $u_{k'}$ cannot be ahead of $u_{k'-1}$. Let s' be maximal (among $s+1, \dots k'-2$) such that $u_{k'}$ is ahead of $u_{s'}$, and let $r' = s+1$. It is easy to check that the sequence $u_{r'}, \dots, u_{k'}$ and s' satisfy the assumptions we made above, and that $k' > k$.

So this construction can be carried out indefinitely. But there are only a finite number of nodes, so we have the desired contradiction to the assumption that any u_k in our sequence can be ahead of u_0. As stated above, this completes the first half of the main induction, since we have shown that if u_0 is stopped before reaching home then it is cancelled within the prescribed time.

There is a rather similar direct proof of the dual result, namely that if u_0 is cancelled before u_0 or its ghost return, then u_0 is stopped before this return. This is not surprising, since in fact it can be shown to follow directly from it by symmetry, as we argue below.

Now that we have restricted ourselves to one update per node, there is a lot of duality in the algorithm. Suppose for a moment that we think of the ring slots as fixed and the nodes as rotating (in the opposite direction to the updates). Further suppose that we identify one slot (real or notional) with each update that will ever be inserted, so that the update actually does fill that slot when inserted. (There will certainly be at least one consistent view of the initial arrangement of these slots on the ring to fulfil this.) Essentially we are thinking of an update's 'ghost' (referred to above) as existing for all time when the update itself does not. Now forget which of the two rings (nodes and updates) is which by identifying live updates with updates expected back by the nodes. It can be seen that the algorithm is completely symmetrical in terms of the two views (in terms of insertion and removal of updates), with stopping in one view corresponding to cancelling in the other. Thus, the fact that we have established the stopping implies cancelling result for u_0 above means that we get this second half by symmetry.

This completes the proof of Lemma 2.2. ∎

Proof of Lemma 2.3 The only way the network could fail to become quiescent would be by an update circulating for ever: necessarily after being cancelled. Lemma 2.2 shows this to be impossible. The only other possible last event other than an update returning home would be some update being stopped at a node N. This is impossible since it would mean that the update generated by N had been stopped but will never be cancelled. Exactly the same argument shows that in the quiescent state there can be no updates expected back. ∎

Proof of Lemma 2.4 Suppose the last update u returns to its origin N, but that some other node N' (which we may assume is, among all such nodes, the one that minimises the distance to N in the direction of the ring) sees activity after u has passed it.

The last activity of N' cannot involve an update generated by any other node, since (i) if that update is retransmitted then some node nearer to N sees it after u, and (ii) if it is stopped at N' then N' is necessarily left expecting its own update back, in contradiction to Lemma 2.3. Thus it must be an update u', generated by N', returning home. By what we know already, neither u nor u' can be ahead of the other. Since they have both returned to their origins it follows that each has passed the other's origin while the latter was expecting its own update. But this is impossible, since one has higher priority which would mean that one is stopped. ∎

We have now proved all that was necessary: that quiescence is reached, that when it occurs all nodes agree on the slot's value, and that all stopping and cancelling occurs within the necessary time. This, with our earlier work justifying the simplifying assumptions, establishes the correctness of the algorithm, or in other words proves Theorem 2.1.

A NOTE ON EFFICIENCY We remarked in describing the earlier algorithm that its throughput declined as the size of the system increased. This need not be true of the current system, since each node can at the limit be receiving and processing updates more-or-less continuously however many nodes there are on the ring; we just need to make sure that the ring has enough slots. Thus the throughput (updates per unit time) is independent of the number of nodes. This is perhaps not as good as we might hope, since more nodes will generate more updates, but this limitation is inherent in the nodes which can process one update at a time as much as it is in the algorithm. A greater criticism of the algorithm might be the high latency (time from an update entering the network to it completing its trip) inherent in the ring topology: this is proportional to the number of nodes. This issue is discussed in more detail in Section 6, where we will see how our algorithms can be adapted for non-rings.

3 A group-like algebra

This section can either be read in advance of, or in conjunction with, Section 4 where a generalised algorithm is introduced which is based on the theory developed here.

The algorithm analysed in the previous section had, like its predecessor, a rather restrictive view of an update: a constant assignment to a single slot. The reason for this was that, as seems inevitable in the class of algorithms we are considering, a pair of updates can reach two nodes in opposite orders. We have seen that it is possible to resolve this conflict by discarding one of a pair of clashing updates providing this simple model is followed; unfortunately if the model of updates is broadened significantly this expedient no longer works.

In the model database system described in the introduction, where each node has a complete copy of the same database, it is reasonable to expect that the model of updates as the assignment of already computed constants will be sufficient in most cases. The exceptions that occur to the author are (i) where the database contains some sort of references between variables, or aliasing, that can be changed by update, (ii) where one does not want to send the constant values because of their large size or for security, and (iii) in applications such as banking, where we would not want only one of a pair of concurrent transactions (say deposits or withdrawals) to take effect. If one adopted a model where not every node held all the data, and where a node might wish to carry out an assignment such as $x := 1 + 2x$ for a variable it did not hold, then things would be different. And it is possible that all the nodes actually hold different databases (where the word 'databases' might now need to be interpreted very liberally), but it is desired that the sequences of updates performed at each of them should be equal (or, at least, have equal effect). Thus it would certainly be of interest to try to broaden the model of updates in our algorithm beyond the constant-assignment one.

The author decided to investigate what would happen if, instead of discarding lower priority updates, a node could modify them to achieve consistency with a node's own update(s), taking into account the fact that some nodes will see one first, some the other(s). The natural equation lying behind this idea is

$$u; v = v; u^v$$

or, in other words, it is possible to *conjugate* any update u by another v so that the effect of

$v; u^v$ is the same as that of $u; v$. The most obvious model for this is in groups (from which the notation is borrowed[4]), where $u^v = v^{-1}uv$, but there is another totally different model based on the one seen in the last section. This is of assignments of values to areas of store: these can be viewed as simultaneous multiple assignments on the one hand, and as functional overrides on the other. If we regard these assignments as sets of pairs $\langle x, a\rangle$, then the effect of an update u is, for all $\langle x, a\rangle \in u$, to modify the value of x in the state to a, and to leave all other values alone. If u is any update, then $dom(u)$ is the set of variables assigned by u. Sequential composition is defined $u; v = v \cup \{\langle x, a\rangle \in u \mid x \notin dom(v)\}$. There are several conjugation operators which satisfy the above equation in this model, the most natural of which identifies u^v with $\{\langle x, a\rangle \in u \mid x \notin dom(v)\}$. (This conjugation operator has the effect of removing from u all assignments to locations which are assigned by v.) Notice that the empty set, corresponding to an assignment to no locations, is a (left and right) identity of ; in this system. The updates of the last section can be thought of as singletons in this system, and a cancelled or stopped update as the empty set (which is the result of conjugating any singleton by a clashing one)[5].

In group theory there are a number of properties that can be derived about the conjugation operator. Since this operator is now being regarded as primitive, some of the ones we require will now have to be stated as axioms. Formally, we will henceforth assume that our updates are drawn from an algebra with two binary operations: $u; v$ and u^v, which satisfy the following laws:

$$\begin{array}{lrcll}
(L1) & u;(v;w) & = & (u;v);w & \text{associativity} \\
(L2) & u;v & = & v;u^v & \text{conjugation} \\
(L3) & (u;v)^w & = & (u^w);(v^w) & \text{distributivity} \\
(L4) & u^{v;w} & = & (u^v)^w & \text{exponentiation}
\end{array}$$

Using laws $L1$ and $L2$ it is easy to prove that $w; (u^w; v^w) = (u; v); w$ and $(v; w); (u^v)^w = u; (v; w)$, but it is not possible to establish the actual identities expressed in $L3$ and $L4$.

We will generally take advantage of $L1$ by omitting bracketing from compositions under ;. Indeed it is convenient to regard two pieces of syntax as equivalent if they only differ in this type of bracketing. (To be completely rigorous we would have to show that all of the later definitions we make recursively on syntax are well-defined under this equivalence. These results are all trivial and most are omitted.)

Since we have $L4$ we will, following the usual mathematical convention, understand u^{v^w} to mean $u^{(v^w)}$. This is because the alternative reading, $(u^v)^w$, is better written $u^{v;w}$.

We will call an algebra which satisfies $L1$-$L4$ a *conjugate algebra*.

The examples mentioned above (groups and multiple assignments) both have identity elements. It is possible (at the price of extra complexity) to do everything we will see in this and the next section without assuming one exists, but since any general language of updates is likely to contain one that has no effect (a null update) there seems little point in paying this price, and in any case it arguably makes for a more pleasing algebra if we have one. Thus we will generally assume our algebra contains a special element 1 (the identity) with the following properties:

$$\begin{array}{lcl}
(L5) & 1;u = u = u;1 & \text{unit} \\
(L6) & 1^u = 1 &
\end{array}$$

[4]It was probably used there because group-theoretic conjugation, like the operators we will use, shares a number of algebraic properties with exponentiation.

[5]Notice that this view of a stopped update puts at least a little flesh on the 'ghosts' introduced in the last section.

A conjugate algebra with an identity will be termed *unitary*. Given a non-unitary algebra it is easy to make it into a unitary one by adjoining an identity element, whose action under the operators is defined by $L5$, $L6$ and $u^1 = u$ (the latter property is implied by $L2$ and $L5$: see below). From here on we will assume our conjugate algebras are unitary unless specifically allowed otherwise.

The reader might like to verify that all of $L1$-$L6$ hold of the multiple assignments example.

There are, of course, numerous identities that can be derived from these laws. Among the most useful are

$$(u^v)^{w^v} = (u^w)^v$$

which comes easily from $L2$ and $L4$, and

$$u^1 = u$$

which follows as $u^1 = 1; u^1 = u; 1 = u$.

Several properties that hold of conjugation in groups do not hold automatically here. A good example is $u^u = u$, which does not hold in the multiple assignments example unless $u = 1$.

The generalised algorithm we present later will be based on updates drawn from a conjugate algebra, with conjugation taking the place of stopping and cancelling. Before coming to this we will develop some properties of our algebra and, in particular, identify some related and sub-algebras which will be useful in analysing the way the algorithm works.

Another obvious example of a conjugate algebra is any commutative monoid (a set with a commutative, associative binary operation with an identity element) with the trivial conjugation $u^v = u$. There are further concrete examples of conjugate algebras and discussion of how they can be combined in Section 5.

Suppose we have a finite set G of constant symbols. Consider the set A_G of expressions built up from these *generators*, together with 1. Given such an expression it is possible to define its *trace*, which is a function from G to the natural numbers $\mathbf{N}$, as follows:

$$\begin{array}{rcll} tr(1)c & = & 0 & \text{for all } c \in G \\ tr(c)d & = & 1 & \text{if } c = d \\ tr(c)d & = & 0 & \text{if } c \neq d \\ tr(e; f)c & = & tr(e)c + tr(f)c & \\ tr(e^f)c & = & tr(e)c & \end{array}$$

This function is defined above on the syntax of expressions, but in fact it is easy to see that the trace is invariant under all of $L1$-$L6$, so that we can think of it as a function on the free algebra generated by G under equality as provable using the laws. The trace of an expression records how many times each generator appears on the 'bottom line', namely not in any exponent. From now on, the usual equality symbol $=$ over A_G will be interpreted as meaning 'provably equal using the axioms' (i.e., equality in the free algebra), while $\equiv$ will mean syntactic equality, modulo the associativity of ;.

Henceforth we will restrict attention to the set S_G of expressions e such that they, and all subexpressions, satisfy $tr(e')c \leq 1$ for all $c \in G$. In other words, no generator appears more than once on the bottom line, so that we can think of the trace as a set. Note that S_G is not closed under ;, so that we shall have to be careful when forming $e; f$ to see that $tr(e) \cap tr(f) = \emptyset$.

We will say that an expression $e \in S_G$ is an *atom* if its trace is a singleton, or in other words if it has the form

$$(\ldots(c^{e_1})^{e_2}\ldots)^{e_n}$$

for some $n \geq 0$, $c \in G$ and $e_1, \ldots, e_n \in S_G$. We say that an expression is *atomic* if it is the sequential composition (;) of 0 or more atoms. (The sequential composition of no things can be identified with 1.)

Every expression in S_G is easily proved equivalent to one in atomic form, and it is useful to have a standard way of generating atomic equivalents.

$$
\begin{array}{rcll}
at(1) & \equiv & 1 & \\
at(c) & \equiv & c & \text{if } c \text{ is an atom} \\
at(e;f) & \equiv & at(e); at(f) & \text{if } at(e) \not\equiv 1 \not\equiv at(f) \\
at(e;f) & \equiv & at(e) & \text{if } at(f) \equiv 1 \\
at(e;f) & \equiv & at(f) & \text{if } at(e) \equiv 1 \\
at(e^f) & \equiv & at(e)^{*f} & \text{if } at(e) \not\equiv 1 \\
at(e^f) & \equiv & 1 & \text{if } at(e) \equiv 1
\end{array}
$$

where, if $e \equiv a_1; \ldots; a_n$ is atomic,

$$e^{*f} \equiv a_1^f; \ldots; a_n^f\,.$$

Notice that, in general, $at(e)$ is atomic, $at(e) = e$ and $at(e) \equiv e$ if and only if e is atomic. The various special cases involving 1 ensure that $at(e)$ is either 1 or the sequential composition of one or more atoms. It is useful to note that $at(e) \equiv 1$ if, and only if, $tr(e) = \emptyset$.

If $a_1; \ldots; a_n$ is atomic then an 'atomic step' is any one of the following manipulations that clearly preserve its atomic nature:

(i) the substitution of any of the exponents in an a_i by an equivalent (i.e., under '$=$') expression;

(ii) law $L4$ applied to one of the a_i or a subexpression of one;

(iii) law $L2$ applied to any consecutive pair of a_i.

If e and f are atomic, and $e = f$ is provable using atomic steps only, then we write $e \stackrel{a}{=} f$.

It is useful to note at this point that the only type of atomic step that does more than change the internal structure of a single atom is (iii).

It is obvious that two atomic expressions are in this relation if, and only if, they can be proved equal using individual applications of the laws which preserve atomicity. It will turn out that atomic expressions are easier to manipulate in some ways than general ones[6]. This concept of equivalence is made important by the following result, which shows that we can restrict proofs of equality of atomic expressions to manipulations within the category of atomic expressions.

Lemma 3.1 If e and f are atomic, then $e = f$ if, and only if, $e \stackrel{a}{=} f$. Thus, for general e and f we have $e = f$ if, and only if, $at(e) \stackrel{a}{=} at(f)$.

Proof It will clearly be enough to prove that, if $e = f$ is provable (for general e and f), using one application of a law, then $at(e) \stackrel{a}{=} at(f)$. We will deal first with the case of a law applied at the outermost level, and will later deal inductively with ones applied in inner contexts. We will deal with the laws one at a time.

Law $L1$ is trivial, since the application of associativity only changes the association of the final atomic forms, which in any case we are ignoring.

[6]This is mainly because one can carry out induction or recursion on their length.

$L2$ is the substantive case. We can assume, without loss of generality, that $e \equiv u; v^u$ and $f \equiv v; u$, where $at(u) \equiv a_1; \ldots; a_n$ and $at(v) \equiv b_1 \ldots b_m$. We will assume that $n, m > 0$; $n = 0$ or $m = 0$ being easy special cases. Then we have

$$\begin{array}{rcll} at(e) & \equiv & at(u); at(v)^{*u} & \\ & \stackrel{a}{=} & at(u); at(v)^{*at(u)} & (1) \\ & \equiv & a_1; \ldots; a_n; b_1^{a_1;\ldots;a_n}; \ldots; b_m^{a_1;\ldots;a_n} & \\ & \stackrel{a}{=} & a_1; \ldots; a_n; (b_1^{a_1;\ldots;a_{n-1}})^{a_n}; \ldots; (b_m^{a_1;\ldots;a_{n-1}})^{a_n} & \\ & \stackrel{a}{=} & a_1; \ldots; a_{n-1}; b_1^{a_1;\ldots;a_{n-1}}; \ldots; b_m^{a_1;\ldots;a_{n-1}}; a_n & (2) \\ & \stackrel{a}{=} & b_1; \ldots; b_m; a_1; \ldots; a_n & (3) \\ & \equiv & at(f) & \end{array}$$

Here, (1) holds because all the manipulations are in exponents, and $v = at(v)$. (2) comes by applying $L2$ m times to commute a_n through to the right hand side. (3) follows by repeating the previous two lines $n-1$ times to move all the other a_i to the right of the b_js.

The remaining laws are all simple to analyse. In the case of $L3$ we may assume that $e \equiv (u; v)^w$ and $f \equiv u^w; v^w$. Assuming that neither $at(u)$ nor $at(v)$ is 1 (which again lead to easy special cases), we get

$$\begin{array}{rcl} at(e) & \equiv & (at(u); at(v))^{*w} \\ & \equiv & at(u)^{*w}; at(v)^{*w} \\ & \equiv & at(f) \end{array}$$

And for $L4$ we may assume $e \equiv u^{v;w}$ and $f \equiv (u^v)^w$.

$$\begin{array}{rcl} at(e) & \equiv & at(u)^{*(v;w)} \\ & \stackrel{a}{=} & (at(u)^{*v})^{*w} \\ & \equiv & at(f) \end{array}$$

(Once again the analysis is slightly different if $at(e) \equiv 1$.) The second line is derived from the first by a number of applications of $L4$ to the individual atoms in the expression (each of which is an atomic step).

The cases of the two laws $L5$ and $L6$ involving 1 are trivial. This completes our analysis of the laws applied at the outermost level. The general case follows by structural induction if we can prove that, on the assumption that $at(u_1) \stackrel{a}{=} at(u_2)$, then all four compositions of the u_i with an arbitrary v produce results whose atomic forms are related by $\stackrel{a}{=}$. The cases where either the u_i or v have atomic form 1 are trivial or make no difference to the analysis, and we will ignore them.

$at(u_1; v) \equiv at(u_1); at(v) \stackrel{a}{=} at(u_2); at(v) \equiv at(u_2; v)$, since the atomic steps executed in the proof of $at(u_1) \stackrel{a}{=} at(u_2)$ can be duplicated. The proof of $at(v; u_1) \stackrel{a}{=} at(v; u_2)$ is exactly the same.

The case of v^{u_i} is trivial, since $at(v^{u_1}) \equiv at(v)^{*u_1} \stackrel{a}{=} at(v)^{*u_2} \equiv at(v^{u_2})$, the centre manipulation coming from the replacements of exponents by their equivalents.

The only mildly difficult case is that of u_i^v. We have assumed that there is a proof in atomic steps of $at(u_1) = at(u_2)$, and must establish that there is also one of $at(u_1)^{*v} = at(u_2)^{*v}$. To convert the first of these proofs into the second, it is clearly enough to demonstrate that, if f is the result of applying a single atomic step to the atomic expression e, then $e^{*v} \stackrel{a}{=} f^{*v}$. We will deal with the different types of atomic steps individually.

The two cases of (i) a manipulation of an exponent and (ii) the application of $L4$ to one of the atoms are both trivial, since exactly the same manipulations of the appropriate component of e^{*v} will produce f^{*v} in a single atomic step.

The final case is of $L2$ applied to a consecutive pair of atoms. To establish this is it sufficient to show that, for a and b atoms, we have $a^v; b^v \stackrel{a}{=} b^v; (a^b)^v$. This is done as follows.

$$\begin{array}{rcl} a^v; b^v & \stackrel{a}{=} & b^v; (a^v)^{b^v} \\ & \stackrel{a}{=} & b^v; a^{v;b^v} \\ & \stackrel{a}{=} & b^v; a^{b;v} \\ & \stackrel{a}{=} & b^v; (a^b)^v \end{array}$$

This completes the proof of Lemma 3.1. ∎

Now suppose that there is a linear order on the set G of generators, representing 'priority'. Consider the subset P_G of S_G with the property that all occurrences of e^f (either at the highest, or at subexpression level), satisfy $u \in tr(e) \wedge v \in tr(f) \Rightarrow u < v$. In other words, one can only conjugate one expression by another of uniformly higher priority. (It will later turn out that this notion of priority is intimately tied up with that used in the first algorithm.) It is easy to see that $at(e) \in P_G$ if $e \in P_G$, and that $e = at(e)$ can proved using the laws without leaving P_G.

In general we will write $e \stackrel{p}{=} f$ if the equality is provable using the laws without leaving[7] P_G, and if they are in addition atomic and the proof was entirely within the atomic members of P_G, we write $e \stackrel{pa}{=} f$. (If an expression is in P_G and is atomic we will in future say that it is P-*atomic*.) Exactly the same analysis as in the proof of Lemma 3.1 yields the following result since, on the assumption that the expressions e and f (equal in one application of a law) are both in P_G, it is easy to establish that the proof we generate of $at(e) \stackrel{a}{=} at(f)$ is also within P_G.

Lemma 3.2 If e, f in P_G are such that $e \stackrel{p}{=} f$, then $at(e) \stackrel{pa}{=} at(f)$. ∎

As if one unusual algebra is not enough, we will now introduce a second one which is derived from the earlier one, but which relates specifically to P_G. We will define an operation $e^{\boxed{f}}$ on P_G for all e, f with disjoint traces. In doing this we will consider the basic equivalence over P_G to be $\stackrel{p}{=}$, so that we will expect, for example, operations to be well-defined under $\stackrel{p}{=}$ but not necessarily under $=$.

We will first define the operation on P-atomic expressions, recursively in their length. This will later be extended to general members of P_G. The unit element behaves in the same way as for the previous operation.

$$1^{\boxed{u}} \equiv u \quad \text{and} \quad u^{\boxed{1}} \equiv u$$

If a and b are atoms, we will write $a < b$ if the (unique) members of their traces are thus ordered. If a and b are atoms, we define

$$a^{\boxed{b}} \equiv \begin{cases} a & \text{if } a > b \\ a^b & \text{otherwise (i.e., } b < a) \end{cases}$$

(Note that our assumption about disjoint traces makes it unnecessary to deal with the case when a and b have the same trace.)

More generally, if $e \equiv e_1; e_2$, then

$$e^{\boxed{f}} \equiv e_1^{\boxed{f}}; e_2^{\boxed{f^{\boxed{e_1}}}}$$

[7] An interesting question that arises from this definition is whether the relation $\stackrel{p}{=}$ is actually different from $=$ over P_G, or in other words whether there is a pair e, f of elements of P_G which are provably equal under $=$, but where no proof is entirely within P_G. The author does not know the answer to this question, but conjectures that the two relations are, in fact, the same.

and, if $f \equiv f_1; f_2$, then

$$e^{\boxed{f}} \equiv (e^{\boxed{f_1}})^{\boxed{f_2}}.$$

The various reduction rules we have given here for computing $e^{\boxed{f}}$ are not disjoint, in the sense that it may be possible to apply them in several different ways to the same expression. This is particularly true when one considers the different (but $\equiv$-equivalent) ways in which a long atomic expression can be bracketed. However, it is a simple induction to show that this operation is well-defined under $\equiv$, so that the order in which the rules are applied does not matter.

If e and f are arbitrary members of P_G with disjoint traces we can now define

$$e^{\boxed{f}} \equiv (at(e))^{\boxed{at(f)}}$$

LEMMA 3.3 The operation $e^{\boxed{f}}$ is well-defined under $\stackrel{p}{=}$, or in other words,

$$e_1 \stackrel{p}{=} e_2 \quad \text{and} \quad f_1 \stackrel{p}{=} f_2 \quad \text{imply} \quad e_1^{\boxed{f_1}} \stackrel{p}{=} e_2^{\boxed{f_2}}.$$

PROOF Since we already know that

$$at(e_1) \stackrel{pa}{=} at(e_2) \quad \text{and} \quad at(f_1) \stackrel{pa}{=} at(f_2)$$

it is sufficient to prove that a single P-atomic step (i.e., an atomic step applied to P-atomic e or f that preserves membership of P_G) does not affect (up to $\stackrel{p}{=}$) the value of $e^{\boxed{f}}$. This is obviously true for the first two types of atomic steps (manipulations of exponents and applications of $L4$), since they only affect one of the atoms in e or f and can be duplicated in the result. Thus all we have to consider is pair swaps.

We first deal with the case where f is an atom and the swap is in e. We will assume that $e \equiv e_1; a; b; e_2$ and prove that if $e' \equiv e_1; b; a^b; e_2$ then $e^{\boxed{f}} \stackrel{p}{=} e'^{\boxed{f}}$. (The cases where either or both of e_1 and e_2 are absent are all easier and extremely similar.) Note that, because $e' \in P_G$, we can assume $b > a$.

A little computation reveals that

$$\begin{array}{rcl} e^{\boxed{f}} & \equiv & e_1^{\boxed{f}}; a^{\boxed{f^{\boxed{e_1}}}}; b^{\boxed{f^{\boxed{e_1;\, a}}}}; e_2^{\boxed{f^{\boxed{e_1;\, a;\, b}}}} \\ e'^{\boxed{f}} & \equiv & e_1^{\boxed{f}}; b^{\boxed{f^{\boxed{e_1}}}}; (a^b)^{\boxed{f^{\boxed{e_1;\, b}}}}; e_2^{\boxed{f^{\boxed{e_1;\, b;\, a^b}}}} \end{array}$$

It should not be hard to see that the equivalence of these two terms will be proved if we can show, for arbitrary atoms c with trace disjoint from those of a and b, that

$$a^{\boxed{c}}; b^{\boxed{c^{\boxed{a}}}} \stackrel{p}{=} b^{\boxed{c}}; (a^b)^{\boxed{c^{\boxed{b}}}}$$

and that

$$c^{\boxed{a;\, b}} \stackrel{p}{=} c^{\boxed{b;\, a^b}}$$

(c is playing the rôle of $f^{\boxed{e_1}}$ in the above expressions.) Given that we know $a < b$ and that traces are disjoint, there remain three possible cases for the order of priority of a, b and c. We will establish the two identities above for each of these.

<u>Case 1: $a < b < c$</u> In this case the second identity is trivial, since both sides equal c. The first becomes

$$a^c; b^c \stackrel{p}{=} b^c; (a^b)^c$$

which is easy, and which has already been dealt with in the proof of Lemma 3.1.

Case 2: $a < c < b$ In this case the second identity is again easy, this time because both sides equal c^b. This time the first becomes

$$a^c; b \stackrel{p}{=} b; (a^b)^{c^b}$$

which follows easily once one notes that $(a^b)^{(c^b)} \stackrel{p}{=} a^{(b;c^b)} \stackrel{p}{=} a^{(c;b)}$

Case 3: $c < a < b$ This time the second identity becomes

$$c^{a;b} \stackrel{p}{=} c^{b;a^b}$$

which is again trivial. This time the first is trivial as well, since it becomes $a; b \stackrel{p}{=} b; a^b$.

(It is interesting to note here that our restriction to P_G and $\stackrel{p}{=}$-equivalence is crucial in the above section of the proof. For if we had not made it we would have had to consider the case where $b < c < a$, which would have resulted in the first identity becoming

$$a; b^{(c^a)} = b^c; a^b$$

which is not true in general[8].)

We can now extend to the case where $f \equiv b_1; \ldots; b_n$ is not an atom by an easy induction: setting $f_r \equiv b_1; \ldots; b_r$, notice that $e^{\boxed{f_r}}$ and $e'^{\boxed{f_r}}$ are both atomic members of P_G. If we assume that they are $\stackrel{pa}{=}$ equivalent, then this equivalence is proved using a number of steps of the types seen above. Since b_{r+1} is an atom it follows from the analysis above of the case where f is an atom that $(e^{\boxed{f_r}})^{\boxed{b_{r+1}}}$ and $(e'^{\boxed{f_r}})^{\boxed{b_{r+1}}}$ are $\stackrel{pa}{=}$ equivalent as well, completing the proof. We have thus dealt with the general case of manipulations to e.

Now consider the case of a manipulation to f. Once again the first two types of atomic step are trivial, and so we can restrict attention to the case where $f \equiv f_1; a; b; f_2$, $f' \equiv f_1; b; a^b; f_2$ and $a < b$.

We must demonstrate the equivalence of the two expressions

$$((e^{\boxed{f_1}})^{\boxed{a;b}})^{\boxed{f_2}} \quad \text{and} \quad ((e^{\boxed{f_1}})^{\boxed{b;a^b}})^{\boxed{f_2}}$$

Since we know (i) that $e^{\boxed{f_1}}$ is P-atomic and (ii) that our result holds for manipulations in the e-argument it will be sufficient to prove

$$d^{\boxed{a;b}} \stackrel{pa}{=} d^{\boxed{b;a^b}}$$

for all P-atomic d with trace disjoint from those of a and b. We already know this when d is an atom, for that was the second identity that we proved in cases in the earlier proof. This can be extended into an inductive proof on the length (i.e., size of trace) of d as follows. If $d \equiv c; d^*$ for some atom c, then

$$\begin{array}{rcll} d^{\boxed{b;a^b}} & \equiv & c^{\boxed{b;a^b}}; d^{*\boxed{(b;a^b)^{\boxed{c}}}} & \\ & \stackrel{pa}{=} & c^{\boxed{a;b}}; d^{*\boxed{(a;b)^{\boxed{c}}}} & \text{by previous case and induction} \\ & & & \text{(see below)} \\ & \equiv & d^{\boxed{a;b}} & \end{array}$$

[8]To see that this identity does not hold, one can either look to the example of groups and ordinary conjugation, or to the area assignment example: in the latter case consider the example where a, b and c are all assignments to the same variable.

The $\stackrel{p}{=}$-equivalence of $d^*\boxed{(b;a^b)\boxed{c}}$ and $d^*\boxed{(a;b)\boxed{c}}$ follows by induction since, in all three cases (of relative priorities), the proof of $\stackrel{p}{=}$-equivalence of $(b;a^b)\boxed{c}$ and $(a;b)\boxed{c}$ produced by our earlier proof consists of a single pair swap plus possibly some type (i) or (ii) P-atomic actions.

This completes the proof of Lemma 3.3. ∎

We have shown that the operation $e\boxed{f}$ is well-defined on P_G for $tr(e) \cap tr(f) = \emptyset$. The new operation is clearly related to, but is subtly different from, the original operation e^f. This relationship is emphasised by the following result, which shows what the new operation's algebraic properties are.

Lemma 3.4 For all e, f and g in P_G with disjoint traces, we have the following:

$P1$. $e; f\boxed{e} \stackrel{p}{=} f; e\boxed{f}$

$P2$. $(e;f)\boxed{g} \stackrel{p}{=} e\boxed{g}; f\boxed{g\boxed{e}}$

$P3$. $e\boxed{f;g} \stackrel{p}{=} (e\boxed{f})\boxed{g}$

$P4$. $1\boxed{e} \stackrel{p}{=} 1$

$P5$. $e\boxed{1} \stackrel{p}{=} e$

Proof $P2$, $P3$, $P4$ and $P5$ come more or less immediately from the definition of $e\boxed{f}$, so we will concentrate on $P1$. It is clearly sufficient to prove it for atomic e and f.

$P1$ holds trivially if e and f are atoms, since one has higher priority than the other, say $e < f$ which means both sides equal $e; f$. If f is an atom then we can prove it by induction on the length of e as follows (noting it is trivially true for $e \equiv 1$). Suppose $e \equiv e'; c$, for c an atom:

$$\begin{array}{rcll} e; f\boxed{e} & \stackrel{p}{=} & c; e'; (f\boxed{c})\boxed{e'} & \\ & \stackrel{p}{=} & c; f\boxed{c}; e'\boxed{f\boxed{c}} & \text{by induction} \\ & \stackrel{p}{=} & f; c\boxed{f}; e'\boxed{f\boxed{c}} & \text{by the above} \\ & \stackrel{p}{=} & f; (c; e')\boxed{f} & \text{by } P2 \end{array}$$

as required, so it is true whenever f is an atom.

We can now complete the proof by an induction on the length of f for general e, noting again that the result holds when $f \equiv 1$. Suppose $f \equiv c; f'$, then:

$$\begin{array}{rcll} e; f\boxed{e} & \equiv & e; (c; f')\boxed{e} & \\ & \stackrel{p}{=} & e; c\boxed{e}; f'\boxed{e\boxed{c}} & \text{by } P2 \\ & \stackrel{p}{=} & c; e\boxed{c}; f'\boxed{e\boxed{c}} & \text{by the above} \\ & \stackrel{p}{=} & c; f'; (e\boxed{c})\boxed{f'} & \text{by induction} \\ & \equiv & f; e\boxed{f} & \end{array}$$

Thus $P1$ holds for all e and f. ∎

To summarise what has been done so far, we have taken one form of algebra, examined a particular subset, P_G, of the words of the free algebra over a given finite set G of generators, and shown that with a stronger notion of equality, $\stackrel{p}{=}$, one can define a new derived operation, $e^{\boxed{f}}$ which has related properties to the first. The precise reasons for us wanting to do these curious things will become apparent in the next section.

Our next step will be to prove a dry-seeming technical result. This will turn out later to be perhaps the key result in the paper, in the sense that it is the one that makes everything else fit together. It is a result about two functions that one can define recursively over a finite set W of expressions from P_G with disjoint traces, which has been endowed with a partial order.

Theorem 3.4 Suppose W is a finite subset of P_G such that (i) if e and f are distinct elements of W then they have disjoint traces and (ii) there is a partial order $\preceq$ on W. Then the pair of functions $f(X)$ (defined for all $X \subseteq W$) and $g(X,a)$ (defined for all $X \subseteq W$ and $a \in W$ such that there is no $b \in X$ with $a \preceq b$) which are defined:

$$\begin{array}{rcll} g(X,a) & \stackrel{p}{=} & 1 & \text{if } b \preceq a \text{ for all } b \in X \\ g(X,a) & \stackrel{p}{=} & g(X\backslash b,a); b^{\boxed{g(x\backslash b,b)}} & \text{otherwise, for } b \text{ any maximal element of } W \text{ which is } \preceq\text{-incomparable to } a \\ & & & \\ f(X) & \stackrel{p}{=} & 1 & \text{if } X = \emptyset \\ f(X) & \stackrel{p}{=} & f(X\backslash a); a^{\boxed{g(X\backslash a,a)}} & \text{otherwise, for } a \text{ any } \preceq\text{-maximal element of } X \end{array}$$

(where $X\backslash a$ is shorthand for $X\backslash\{a\}$) are well-defined. (This is an issue because the two main clauses each select one of a class of maximal elements. It is by no means obvious that the value of the function is independent of which one is selected.)

Proof Since the definition of f uses g, we will tackle g first. The proof will be by induction on the size of X, the result being trivial for $X = \emptyset$, since then $g(X,a) = 1$. So suppose it holds of all sets smaller than X. By this assumption all the recursive calls that are made on the definition of g other than the one at the highest level in computing $g(X,a)$ are known to be well-defined. It follows that the only possible source of a problem would be the existence of maximal b, c in X, each incomparable to a, such that

$$g(X\backslash b,a); b^{\boxed{g(X\backslash b,b)}} \stackrel{p}{\neq} g(X\backslash c,a); c^{\boxed{g(X\backslash c,c)}}$$

(the inductive assumption means that all these instances of g are well-defined). Note that b and c are necessarily incomparable, and that b is maximal in $X\backslash c$ and *vice versa*. It follows that the left-hand-side of the above inequality can be shown equivalent to

$$g(X\backslash\{b,c\},a); c^{\boxed{g(X\backslash\{b,c\},c)}}; b^{\boxed{g(X\backslash\{b,c\},b); c^{\boxed{g(X\backslash\{b,c\},c)}}}}$$

by expanding the two calls of g. $P3$ applied to the last term converts this to

$$g(X\backslash\{b,c\},a); c^{\boxed{g(X\backslash\{b,c\},c)}}; (b^{\boxed{g(X\backslash\{b,c\},b)}})^{\boxed{c^{\boxed{g(X\backslash\{b,c\},c)}}}}$$

Now notice that $P1$ is applicable to the last two terms, and converts it to the term

$$g(X\backslash\{b,c\},a); b^{\boxed{g(X\backslash\{b,c\},b)}}; (c^{\boxed{g(X\backslash\{b,c\},c)}})^{\boxed{b^{\boxed{g(X\backslash\{b,c\},b)}}}}$$

which is symmetrical to it under the exchange of b and c. This symmetry (or working the original analysis backwards with b and c swapped) means that it equals $g(X\backslash c, a); c^{\boxed{g(X\backslash c, c)}}$, the right-hand side of the inequality, a contradiction. We can infer that g is, indeed, well-defined.

A very similar argument now demonstrates the well-definedness of f. It is clearly well-defined for $X = \emptyset$, and so, on the assumption that f is not well-defined, there is a smallest and non-empty X for which it fails. Much as in the case of g, the only way this ill-definedness can arise in this minimal X is for there to be incomparable, maximal a, b, such that

$$f(X\backslash a); a^{\boxed{g(X\backslash a, a)}} \overset{p}{\not\equiv} f(X\backslash b); b^{\boxed{g(X\backslash b, b)}}$$

(where all the subterms are well-defined). Observe that b is maximal in $X\backslash a$, and a in $X\backslash b$. We again expand the left-hand-side, this time obtaining

$$f(X\backslash\{a,b\}); b^{\boxed{g(X\backslash\{a,b\},b)}}; a^{\boxed{g(X\backslash\{a,b\},a); b^{\boxed{g(X\backslash\{a,b\},b)}}}}$$

which law $P3$ converts to

$$f(X\backslash\{a,b\}); b^{\boxed{g(X\backslash\{a,b\},b)}}; (a^{\boxed{g(X\backslash\{a,b\},a)}})^{\boxed{b^{\boxed{g(X\backslash\{a,b\},b)}}}}$$

As in the case of g, law $P1$ is applicable to the last two terms to obtain

$$f(X\backslash\{a,b\}); a^{\boxed{g(X\backslash\{a,b\},a)}}; (b^{\boxed{g(X\backslash\{a,b\},b)}})^{\boxed{a^{\boxed{g(X\backslash\{a,b\},b)}}}}$$

which is symmetric with it under the interchange of a and b. As in the case of g, this contradicts our assumption of f's ill-definedness, completing the proof of the theorem. ∎

Up to this point we have been a little schizophrenic in our treatment of the algebra of the box-conjugation operator $a^{\boxed{b}}$: it has simultaneously been an algebra in own right, governed by the laws $L1$ and $P1$-$P5$, and also a way of investigating properties of the earlier one. For the rest of this section we will concentrate solely on the former, and call this algebra the *box algebra*.

The definitions of f and g above can be unwound (at least for partial orders with many incomparable elements) in many different ways, which we have now proved to be equivalent. We will call one of the ways in which $f(X)$ or $g(X, a)$ can be written down a *presentation*. Just how many there can be is illustrated by considering the worst case, which is of the completely flat partial order Z_n with n elements, where no pair is ordered. The number, t_n, of different presentations of $f(Z_n)$ can be computed by the recurrence

$$t_1 = 1 \qquad t_{n+1} = (n+1) \times t_n^2$$

which leads to the truely formidable number

$$t_n = n(n-1)^2(n-2)^4 \ldots 2^{2^{n-2}}$$

which grows as an exponential of an exponential.

It is possible to show that the function f captures the structure of the partial order $\preceq$ completely. For suppose that the set W over which we are defining f and g is G, the set of generators[9].

[9] In fact, for the result discussed here to hold it is sufficient that no element of W has an empty trace. The more general argument would, however, simply complicate matters for no real gain.

Then, if $\preceq_1$ and $\preceq_2$ are two partial orders on G and f_1, f_2, g_1 and g_2 are defined using the respective orders, we can show that the expressions $f_1(G)$ and $f_2(G)$ are provably equal if, and only if, the orders are in fact the same. In order to prove this we will first establish the following.

LEMMA 3.5 If $W = G$ in the above definitions of f and g, and it can be proved using $P1$-$P5$ (plus $L1$) that $f(X) = e; a^{\boxed{f}}$ for any $a \in X$ and expressions e and f, then a is maximal in X and e, f are provably equal to $f(X\backslash a)$ and $g(X\backslash a, a)$ respectively. (In other words, our definition of f earlier gives all interesting ways of writing $f(X)$ down.) Similarly, if $g(X, a)$ can be proved equal to $e; b^{\boxed{f}}$ then b is maximal in X, incomparable to a, and e, f are provably equal to $g(X\backslash b, a)$ and $b^{\boxed{g(X\backslash b, b)}}$ respectively.

PROOF It is easy to redefine the concepts of 'atom' and 'atomic expression' in the box algebra, and to devise a standard way of reducing every expression to atomic form. Clearly all presentations of $f(X)$ and $g(X, a)$ are atomic. Essentially the same analysis as was carried out for conjugate algebras will show that two atomic expressions of the box algebra are provably equal if and only if they can be proved equal in atomic steps (which again are internal manipulations of a single atom or an adjacent pair swap, this time using $P1$). It follows that we can assume that the expressions e and f in the statement of the Lemma are in atomic form, and that the proof of equality with $f(X)$ or $g(X, a)$ is in atomic steps.

As before we will deal with the case of g first, and work by induction on the size of X. Consider the class of L-equivalents of presentations of $g(X, a)$: expressions which can be obtained from such a presentation by applying atomic steps which alter the structure of individual atoms. (The L here stands for local.) We will clearly have established the result for X if we can show that this class is closed under pair swap atomic steps: this closure condition, in fact, will be our inductive hypothesis. The result is trivial when $X = \emptyset$, so suppose it holds of all smaller sets than X. Any member of the class clearly has the form

$$e; (b^{\boxed{f_1}} \ldots)^{\boxed{f_n}}$$

where b is incomparable to a and maximal in X, e an L-equivalent of some presentation of $g(X\backslash b, a)$ and $f_1; \ldots; f_n$ is equivalent to $g(X\backslash b, b)$. If a pair swap does not involve the last two terms then we can appeal to induction. So suppose it is a swap of the last two terms; we know that e has the form $e'; f'$ for some e' equivalent to $g(X\backslash\{b, c\}, a)$ and f' equivalent to $c^{\boxed{g(X\backslash\{b,c\},c)}}$ for c maximal in $X\backslash b$ and incomparable to a. For this swap to be possible, it follows that f_n is same as f', which in turn means that $g(X\backslash b, b)$ is equivalent to something of the form $h; c^{\boxed{g(X\backslash\{b,c\},c)}}$. By induction applied to the term $g(X\backslash b, b)$, this is an L-equivalent of some presentation of $g(X\backslash b, b)$, and so c must be incomparable with b as well as a. And h (which is equivalent to $f_1; \ldots f_{n-1}$) is now also known to be equivalent to $g(X\backslash\{b, c\}, b)$. When the pair swap has been carried out we get

$$e'; (b^{\boxed{f_1}} \ldots)^{\boxed{f_{n-1}}}; f'^{\boxed{(b^{\boxed{f_1}} \ldots)^{\boxed{f_{n-1}}}}}$$

which is L-equivalent to

$$g(X\backslash\{b, c\}, a); b^{\boxed{g(X\backslash\{b,c\},b)}}; c^{\boxed{g(X\backslash\{b,c\},c); b^{\boxed{g(X\backslash\{b,c\},b)}}}}$$

which, since c is (given what we already know about it) necessarily maximal in X, is a presentation of $g(X, a)$.

The analysis of f is very similar: one again shows inductively that the class of L-equivalents of presentations are closed under pair swaps, this time using the above result to deal with calls of g. ∎

We are now in a position to prove the following result.

THEOREM 3.6 If $W = G$ then the value of $f(W)$ in the box algebra completely characterises the partial order $\preceq$ used to construct f. In other words, if f_1, g_1, f_2 and g_2 are constructed using partial orders $\preceq_1$ and $\preceq_2$ over the same W, then $f_1(W)$ is equivalent to $f_2(W)$ if, and only if, $\preceq_1=\preceq_2$.

PROOF The 'if' of this result is trivial. We will prove that $f_1(W) = f_2(W)$ implies equality of orders by induction on the size of W. The case of $|W| = 1$ is trivial, so suppose it holds of all smaller W and that $f_1(W) = f_2(W)$. Consider a typical presentation of the right hand side:

$$f_2(W\backslash a); a^{\boxed{g_2(W\backslash a, a)}}$$

for a $\preceq_2$-maximal in W. Because we are assuming that this is provably equal to the left hand side, and because of the result of Lemma 3.5, we know that a is maximal with respect to $\preceq_1$ also, that $f_2(W\backslash a)$ equals $f_1(W\backslash a)$ and that $g_2(W\backslash a, a)$ equals $g_1(W\backslash a, a)$. Induction then tells us that the two orders coincide on $W\backslash a$. But the equality of the g_i on $(W\backslash a, a)$ show that precisely the same set of generators are less than or equal to a in the two orders since it is easy to see that the trace of $g(W, a)$ contains, in general, precisely those elements of W which are not less than a. Since a is already known to be maximal in both orders it follows that the two orders coincide on the whole of W, as required. ∎

Theorems 3.4 and 3.6 together show that the functions f and g in some sense allow us to capture partial orders algebraically. It seems likely to the author that this work will have applications beyond those on databases described in the rest of this paper: perhaps to the partially ordered computation histories found in the theory of 'true concurrency'. Our algebra provides an interesting contrast to Mazurkiewicz traces [M], where a different monoid is used to describe each dependence relation. The free box algebra over a set of generators provides a single (though more complex) structure over which any such relation can be expressed.

As discussed in the introduction, it has turned out that the concept of a box algebra is essentially the same as that of Gene Stark's 'algebra of residuals' [S], which he introduced specifically to reason about true concurrency. His results are rather different from ours, because of the way in which the algebra arose in his work.

4 Algorithm 2: algebraic updates

The work of the previous two sections can be put together to produce a ring-based algorithm that works for any language of updates satisfying laws $L1$-$L6$. The basics of the algorithm are similar: we once again base it on a no-overtaking ring (assumed to have the same basic properties) and give each node queues Q_i of updates executed locally but not yet inserted into the ring, and E_i of the updates it is expecting back from the ring. And again each node is assumed to have a unique priority p_i.

The updates that nodes generate are assumed to be drawn from an algebra of the type seen at the last section. We have already seen two examples of algebras satisfying the necessary axioms (multiple assignments and groups), plus the trivial case of a commutative monoid. There will be some discussion of other models in the next section. The various possible actions of a node are described below. Notice that each update now circles the system exactly once: conjugation takes the place of stopping, and of cancelling too.

Even though an update may have been altered (i.e., conjugated) since it was originally generated, its priority never changes, always being that of its originator.

- If N_i generates an update u, then u is executed locally and u is inserted at the tail of Q_i.
- If Q_i is nonempty and there is a space available on the ring, then the head of Q_i is removed and inserted into the ring, and into the tail of E_i.
- If an update u' returns which was originated by N_i as u (but may have been altered since), then it is removed from the ring, and from the head of E_i.[10]
- If an update u arrives that originated at N_j with $p_j < p_i$, then $u^{E_i;Q_i}$ is executed locally and u^{E_i} is passed on round the ring. E_i is unaltered.
- If an update u arrives that originated at N_j with $p_j > p_i$, then u^{Q_i} is executed locally, u is passed on unaltered round the ring and E_i is replaced by E_i^u (i.e., each element of E_i is conjugated by u).

The way we are using conjugation here is always to compensate for different nodes seeing updates in different orders. In replacing u by u^v we are compensating for the fact that later nodes will see u after v, even though some others are seeing it before v. The clearest way to see this at a simple level is to examine what happens when two nodes inject updates into the system at the same time (and these are the only updates in circulation).

The careful reader might have noticed one difference with Algorithm 1, relating to our treatment of updates in Q_i. In the first algorithm we deleted updates from Q_i where they clashed with u, while in this one we have conjugated u by Q_i (i.e., the exact opposite). In fact we *could* have gone either way round in either case. We took the approach we did in Algorithm 1 because it cut down the number of updates in circulation, as well as for similarity with the commercial approach. We have chosen here to do the opposite both to illustrate that it can be done and because it seems more natural that updates inserted later into the system should be viewed by the system as having been executed later. And here, chosing the other option would not cut down the number of updates to be transported.

The main theorem of this section is the following.

Theorem 4.1 If all copies of the database were equal initially and the above algorithm is used for updates, then when it becomes quiescent (no updates queued in any Q_i or circulating) then all copies will still be equal. Or equivalently: when the system becomes quiescent then the updates executed at each node are provably equivalent. ∎

Most of the rest of this section will be devoted to proving this result. However we will first give an example to illustrate how the algorithm works.

Example We will illustrate the operation of the algorithm with an example where there are five nodes, each generating one update. Suppose they are arranged round the ring clockwise, the direction in which updates are carried, in the order N_1, N_4, N_2, N_5, N_3, where, the lower the index, the higher a node's priority. The updates generated by $N_1, \ldots, N_5$ are $a, \ldots, e$ respectively. Suppose a and b are emitted at time T_1, and that the other three are emitted at time T_2, by which time a has passed N_4 and b has passed N_5 and N_3. For simplicity we will suppose that no update waits in a Q_i: there is always immediate space in the ring. No conjugation will have occurred by time T_2, since every time an update reached a node the latter's E_i was empty. The sequence of

[10] In this algorithm it is clear that at this moment the message now sitting at the head of E_i is what has become of the the copy of u originally placed there when u started its journey. However it is by no means obvious that the two copies of u have been transformed to the same thing. It is true though, as we shall see later. However, for now, it is perhaps better to state simply that u' and the head of E' are both removed.

updates seen by each node is now determined, as are the values of the E_i seen by the updates as they travel round. We can track the progress of the algorithm as follows:

- A little later than T_2, b and then c may have passed N_1, becoming conjugated to b^a and c^a respectively. a may have passed N_2, conjugating E_2 to b^a, followed by d, which becomes conjugated to d^{b^a}. e may have passed N_3, becoming conjugated to e^c.
- Yet later, b^a and c^a will pass N_4, conjugating E_4 to $d^{(b^a);(c^a)}$. a and d^{b^a} will pass N_5, conjugating E_5 to $e^{a;(d^{b^a})}$. e^c may now have passed N_1, becoming conjugated to $e^{c;a}$.
- b^a will then return to N_2, cancelling E_2, so that when c^a passes N_2 it is not conjugated. It does, however, conjugate E_5 when it passes to $e^{a;(d^{b^a});c^a}$ before returning home to N_3. a will pass N_3, conjugating E_3 to c^a before returning home itself. d^{b^a} will be conjugated at N_3 to $d^{(b^a);(c^a)}$, but not at N_1. $e^{c;a}$ is conjugated at N_4 but not at N_2, arriving home as the complex term $e^{c;a;(d^{(b^a);(c^a)})}$.

It is interesting to tabulate both the sequence of updates that have been executed at the various nodes, and to compare the values of the updates that return home with the values that were waiting for them in the E_i. We tabulate these second things first. The left-hand column contains the value of the update that returned, the right hand one the value that was waiting.

$$\begin{array}{cc}
a & a \\
b^a & b^a \\
c^a & c^a \\
d^{(b^a);(c^a)} & d^{(b^a);(c^a)} \\
e^{c;a;(d^{(b^a);(c^a)})} & e^{a;d^{b^a};(c^a)}
\end{array}$$

These are all obviously equal except for the last pair, the exponents of which are easily proved equal by pushing c through a in the left hand side to get $a; c^a; (d^{b^a})^{c^a}$ and then applying $L1$ to get the right hand side. Thus, in this case at least, the updates that return are equal to those that are expected. The updates actually executed at the nodes are as follows.

N_1: $a; b^a; c^a; e^{c;a}; d^{(b^a);(c^a)}$

N_2: $b; a; d^{b^a}; c^a; e^{c;a;(d^{b^a};c^a)}$

N_3: $b; c; e^c; a; d^{(b^a);(c^a)}$

N_4: $a; d; b^a; c^a; e^{c;a;(d^{b^a};c^a)}$

N_5: $b; e; a; d^{b^a}; c^a$

All of these expressions are superficially different, but a little algebra proves them equal: expression 1 is equal to expression 3 once one applies $L2$ three times to push the a three places to the right. This, in turn equals

$$(*) \qquad b; e; c; a; d^{(b^a);(c^a)}$$

by applying $L2$ to $c; e^c$. Pushing the c one place to the right in this we obtain

$$b; e; a; c^a; (d^{b^a})^{c^a}$$

which $L2$ shows equal to expression 5. If we push e all the way through to the right in $(*)$ we get

$$b; c; a; d^{(b^a);(c^a)}; e^{c;a;(d^{b^a};c^a)}$$

which we can manipulate to expression 2 by pushing c to the right of the a to get

$$b; a; c^a; d^{(b^a);(c^a)}; e^{c;a;(d^{b^a};c^a)}$$

and then applying $L2$. We can get expression 4 from expression 2 by pushing a to the right of the b, getting

$$a; b^a; d^{b^a}; c^a; e^{c;a;(d^{b^a};c^a)}$$

and then applying $L2$ to the second and third terms. Hence they are all equal. It is interesting to note (and this will later be significant) that in none of the above proofs was there ever an instance of a high priority term being conjugated by a low priority one. In the language of the last section, all expressions were in P_G, where $G = \{a, b, c, d, e\}$. ∎

As in the case of the first algorithm, it is useful to make some simplifying assumptions in the proof. Firstly, we would again like to be able to assume that the Q_i's are absent. Secondly, we would like to be able to restrict attention to the case where each node is limited to emitting one update. (With our generalised concept of update it no longer makes sense to talk about restricting to a single slot.) The justifications of these assumptions are given in the next two lemmas (and are very similar to the corresponding justifications in the proof of the first algorithm).

Lemma 4.2 In any given finite execution of a ring running Algorithm 2, the total effect of all the updates executed at a node is the same as would be achieved by (i) delaying the local execution of each update until its insertion into the ring and (ii) omitting to conjugate incoming updates by Q before local execution.

Proof As in the case of Algorithm 1, this manipulation in no way affects either the pattern of updates actually circulating in the ring, or the content of the expected-back queues E_i. Thus the sequence of updates which arrive at a given node from the ring is unaffected.

Suppose the last local update generated by a node N is u. Then the sequence of all updates executed at N in the original algorithm might be

$$v_0; \ldots; v_k; u; w_1^u; \ldots; w_n^u; t_1; \ldots; t_m$$

Here, the v_i are the updates executed at N before u is generated. Updates w_i' arrive while u is in Q and are conjugated to w_i before their final conjugation by u (which is last on Q throughout this time). The t_i are the updates executed after u enters the ring. Under our algebra the above sequence is equal to

$$v_0; \ldots; v_k; w_1; \ldots; w_n; u; t_1; \ldots; t_m$$

which is exectly the updates that would have been executed if the generation of u had been delayed until its insertion into the ring. This construction can clearly be repeated to move the generation of all of N's updates to the point of their insertion into the ring, and the final sequence of updates obtained will be exactly the one described in the statement of the Lemma. ∎

Henceforth we will restrict our attention to systems where there are no Q_i.

Lemma 4.3 Theorem 4.1 holds on the assumption that it holds of all systems where no node emits more that one update.

Proof The proof of this result is essentially the same as that for the corresponding assumption under Algorithm 1. In an execution of the system where no node emits more than r updates we can replace each node by a sequence of r consecutive ones. As before, the priorities are small perturbations of those of the node they replace. Exactly the same construction as before works, restricting each sequence of nodes to dealing with one update at a time. Each update that arrives is either conjugated by the contents of the $E_{i,j}$ in the new system, or conjugates them, in a way exactly analogous to what happens in the original system. (It is important in this to note that any update from a different sequence that arrives is either of lesser priority than all the $N_{i,j}$, or of greater priority than them all). Once again, the last member $N_{i,r}$ of the sequence sees exactly the same sequence of updates as is seen by N_i in the original system. ∎

It will thus be sufficient to prove Theorem 4.1 for the case of one update per node. We can place a partial order on the updates which arise in a given execution (quite independently of the priority that already exists) as follows: $u \prec v$ if u passes the origin of v before v is emitted. The no-overtaking nature of the ring guarantees that this is a partial order, and that $u \prec v$ if and only if u arrives[11] at all nodes before v does. (For, if $u \not\prec v$ then v arrives at u's origin before u arrives back there.) If u and v are incomparable under $\prec$ then we write $u\|v$ and observe that this happens precisely when some points on the ring have u arrive first, and some have v. (To be specific, the nodes on the ring from the one after N_u to N_v, inclusive, have u arrive first, and the rest have v arrive first.)

Consider the free conjugate algebra over a set of generators containing one element for each update: we will consider the operation of the algorithm when each node, at its appointed moment, emits the appropriate generator as its update. Our objective is to prove, in the free algebra, that the effect of the updates arriving at each node in this situation are equal: for if we can do this then the result will remain true if the generators are replaced with the actual values of updates drawn from any conjugate algebra. Observe that the value of any update as it passes round the ring remains an atom whose trace element is the generator it started as, and that, under our assumptions, each expected back list E_i is either empty or contains a single atom.

Using the priority order which the updates inherit from their originators, we can make the free algebra into a box algebra as described in the last section. The reason why this extra algebra was introduced in the last section can now be made apparent. For recall that the operator $a^{\boxed{b}}$ was defined for atoms:

$$a^{\boxed{b}} = \begin{cases} a & \text{if } a > b \\ a^b & \text{if } a < b \end{cases}$$

Noticing that, in the simplified case we are now considering, the only conjugation which takes place in our algorithm is between atoms, the two central clauses of the algorithm (which govern conjugation) can now be replaced by the following, in which the list E of the original algorithm has been replaced by a single value e: 1 when E is empty, and otherwise the single element of E. (Notice that when conjugation takes place in the algorithm, it is always between terms with disjoint traces.)

- If an update u arrives, which originated elsewhere, then $u^{\boxed{e}}$ is executed and passed on, and e is replaced by $e^{\boxed{u}}$.

The beauty of this is that the asymmetry of priority has been hidden, which means that it is not necessary to worry in our proof about the $(n-1)!$ different ways of arranging the order of priorities around the ring. Furthermore, all nodes now look exactly the same.

[11] We have used the word 'arrives' with care here: we consider the moment when an update arrives at its origin to be when it returns, not when it sets off.

We now claim that, with respect to this box algebra and the partial order $\prec$ (on the set U of generators):

1. the updates executed over the run of the algorithm at each node equal $f(U)$, and
2. when an update (originally the generator a) arrives back at its origin N_a, both it and the expected-back version equal

$$a^{\boxed{g(\{b \in U \mid a \not\preceq b\}, a)}}$$

The first of these claims, when established, will prove Theorem 4.1. The second justifies the claim that, when an update arrives back home, *it* is removed from E. These facts show why we were so interested in the functions f and g in the last section. As has been seen in the five-node example given earlier, and may have been discovered by the reader in experimenting, our algorithm can discover a wide variety of ways of writing down expressions that are meant to be equivalent. It turns out that the well-definedness proofs of f and g give the ideal ways to prove that they always are equivalent. In order to justify these claims we will prove the following lemma, which generalises the second claim.

LEMMA 4.4 When an update (emitted as the generator b) arrives at a node N, its value is

$$b^{\boxed{g(k(b,N),b)}}$$

and, if N has an expected-back update that was emitted as (the generator) c, then at the moment b arrives the expected-back value is

$$c^{\boxed{g(k(b,N),c)}}$$

where $k(b,N) = \{a \in U \mid a \prec b \vee (a \| b \wedge N_a \in (N_b, N))\}$ (N_a is the origin of an update a and (N, M) is the set of nodes strictly between N and M on the ring, in the direction in which updates are transmitted.)

PROOF It should not be too hard to see that the set $k(b,N)$ consists of those updates which arrive at N before b does. If $N = N_b$ then $k(b,N)$ becomes exactly the set which appears in the second claim above.

We prove the Lemma by induction on the number of events of the form 'update a arrives at node M' which have occurred before the given one.

If b has not yet visited any 'expectant' nodes on its travels, then its value is still b. This is the value predicted by the Lemma since the set $\{a \mid b \| a \wedge N_a \in (N_b, N)\}$ is empty, which means that $k(b,N)$ contains only values strictly less than b. Otherwise there is a last expectant node – N_d, say – which b has visited. Clearly $k(b,N) = k(b,N_d) \cup \{d\}$ by construction of k. Since d is the nearest element of $\{a \mid b \| a \wedge N_a \in (N_b, N)\}$ to N, it must be maximal in $k(b,N)$: if d had passed any other N_c for $c \in k(b,N)$ before c were emitted then d must also pass N_b before b is emitted – a contradiction. We know, by induction, that when b arrived at N_d, their respective values were

$$b^{\boxed{g(k(b,N_d),b)}} \qquad \text{and} \qquad d^{\boxed{g(k(b,N_d),d)}}$$

so it follows that the value of b when it arrives at N is

$$b^{\boxed{g(k(b,N_d),b);\, d^{\boxed{g(k(b,N_d),d)}}}} = b^{\boxed{g(k(b,N),b)}}$$

as required.

Now suppose N has an outstanding update, which was originally c. If no updates have arrived at N between c arising and b arriving, then its value is still c when b arrives. This is the value predicted by the Lemma, since under this assumption all elements a of $k(b, N)$ (the updates that arrive at N before b) must have arrived at N before c arose and hence satisfy $a \prec c$.

Otherwise there was a last update, originally d, say, which arrived at N after c arose but before b arrived. Necessarily $d \| c$. Clearly $k(b, N) = k(d, N) \cup \{d\}$ thanks to our characterisation of $k(a, N)$ as the set of updates that arrive at N before a. Furthermore, d is maximal in $k(b, N)$ since N sees d after all other elements of that set.

By induction, the values of d and the expected-back copy of c were, just before d arrived at N,

$$d^{\boxed{g(k(d,N),d)}} \qquad \text{and} \qquad c^{\boxed{g(k(d,N),c)}}$$

When d arrives at c, the expected-back value of c is thus conjugated to

$$c^{\boxed{g(k(d,N),c);\, d^{\boxed{g(k(d,N),d)}}}} = c^{\boxed{g(k(b,N),c)}}$$

by definition of g and what was established above. Since this is the value that is expected when b arrives, and is what the Lemma predicts, this completes the proof of the Lemma. ∎

This establishes our second claim and puts us in a position to establish the first, which is a corollary to the following lemma.

LEMMA 4.5 We know that all updates in U are executed at all nodes N, so suppose $\{a_1, \ldots, a_n\}$ enumerates U in the order in which the updates are executed at a given N. Then the effect of the first r updates $(0 \leq r \leq n)$ executed at N is precisely $f\{a_1, \ldots, a_r\}$.

PROOF Suppose that the update emitted by N is a. (If there is no such update, then the proof is a trivial simplification of what follows: only case (ii) below applies.) Observe that, when an update a_r arrives at N which originated elsewhere, then

$$\{a_1, \ldots, a_{r-1}\} = \begin{cases} k(a_r, N) & \text{if } a \text{ is not then expected back} \\ k(a_r, N) \cup \{a\} & (\text{and } a \notin k(a_r, N)) \text{ otherwise} \end{cases}$$

since a is executed when first emitted, not when it arrives back.

We will prove the Lemma by induction on r. (For all N simultaneously.) It certainly holds before any updates are executed, since $f(\emptyset) = 1$. So suppose it holds for $r - 1$. There are three cases to consider when considering the execution of a_r.

<u>Case (i): $a_r = a$</u> In this case, clearly $a_i \prec a_r$ for all $i < r$ since all such a_i necessarily pass N before a is emitted. It follows that $g(\{a_1, \ldots, a_{r-1}\}, a_r) = 1$, which in turn implies that

$$f\{a_1, \ldots, a_r\} = f\{a_1, \ldots, a_{r-1}\}; a_r \,.$$

This, by induction, equals the actual sequence of updates executed up to a_r (since N executes its own update unaltered at the same time as it enters the ring).

<u>Case (ii): $a_r \neq a$ and a is not expected when a_r arrives</u> In this case, as we have observed, $\{a_1, \ldots, a_{r-1}\} = k(a_r, N)$, and since a_r is plainly $\prec$-maximal in $\{a_1, \ldots, a_r\}$ (N sees it last of this set), we have

$$f\{a_1, \ldots, a_r\} = f\{a_1, \ldots, a_{r-1}\}; a_r^{\boxed{g(k(a_r,N),a_r)}}$$

which, by induction and Lemma 4.4 applied to a_r's value on arrival at N, is exactly the sequence of updates executed at N up to a_r.

Case (iii): $a_r \neq a$ and a is expected when a_r arrives In this last case, we have observed that $\{a_1, \ldots, a_{r-1}\} = k(a_r, N) \cup \{a\}$, and a, a_r are both $\prec$-maximal in $\{a_1, \ldots, a_r\}$. For the last update to arrive at N from this set is a, and the last one from this set to arrive at the node after N (in the direction of the ring) is a_r. From the definition of f we get

$$f\{a_1, \ldots, a_r\} = f\{a_1, \ldots, a_{r-1}\}; a_r^{\boxed{g(\{a_1, \ldots, a_{r-1}\}, a_r)}}$$

which the definition of g rewrites to

$$f\{a_1, \ldots, a_{r-1}\}; a_r^{\boxed{g(k(a_r, N), a_r); a^{\boxed{g(k(N, a_r), a)}}}}$$

We know that, when a_r arrives at N, the value of a_r and the expected-back value of a are respectively

$$a_r^{\boxed{g(k(a_r, N), a_r)}} \qquad \text{and} \qquad a^{\boxed{g(k(a_r, N), a)}}$$

which, since the a_r that arrives is conjugated before execution by the current value of a, tells us that, by induction, the updates executed up to a_r are equivalent to (the above expansion of) $f\{a_1, \ldots, a_r\}$, as required. ∎

This completes our proof of Theorem 4.1. It is perhaps worth reflecting briefly on our proof. What we were interested in doing was proving that, whatever the arrangement of node priorities round the ring, and whatever the order in which the updates were inserted, the updates executed at each node were provably equivalent. Now 'provably equivalent' in this context meant the same as saying that the sequences of updates were equal in the free algebra where each new update was a different generator. We overcame the complexity of the arrangement of nodes by moving to the box algebra, where the algorithm became symmetric. Even though the different nodes might see the updates in very different orders, the orders are always consistent with the natural partial order $\prec$ which describes which pairs of updates are ordered from all viewpoints. And it turned out that the symmetrised algorithm is simply computing the function $f(U)$ for all nodes, using the presentations corresponding to the orders in which the various nodes see the updates.

The well-definedness proofs of f and g break up the inductions required to prove the main result down into manageable size pieces. If one attempts a more direct induction without using these functions, the problem suddenly seems much less structured.

It is interesting to note that we only managed to obtain our proof of the fact that the expressions were provably equal by effectively tieing one hand behind our back: for we limited ourselves to proofs in which all intermediate expressions lay in P_G (i.e., where no low-priority expression ever conjugates a high-priority one). By doing this we managed to derive the box algebra and its corresponding theorems, even though we were under no obligation, given the way our algorithm was set up, to limit ourselves to proofs of this form.

We saw earlier how Algorithm 2 generalises Algorithm 1. (The fact that one algorithm stops or cancels messages altogether, while its translation conjugates them to 1 instead can be disregarded for our purposes, since conjugation by or of 1 has no effect. Our result about the returning update being the same as the one which is expected shows that an update is stopped if and only if cancelled, etc.) Therefore, in some way, our proof of Algorithm 2 must contain another proof of Algorithm 1. Since, stylistically, our two proofs have been very different, it might be an interesting exercise to see just how the proof in Section 2 and the proof of Algorithm 1 that one can derive relate to each other.

4.1 Variants of the algorithm; timestamping

Throughout our descriptions of Algorithms 1 and 2 we have derived the priority of updates from (relatively) fixed node priorities. However there is no particular reason why the priorities have to be assigned in this way, although we do have to be a little more careful in our description of the algorithm if the priority mechanism is relaxed. The key to seeing how the algorithm should work in a more general setting, where all updates that might be concurrent are given distinct priorities, can be found in the technique we evolved in our two proofs of splitting nodes into units which emit one update each. For of course if this restriction is observed then we cannot tell between our original algorithms and the ones where priorities are not linked to nodes. What a single node which emits several updates must do is emulate the behaviour of a sequence of single-update nodes with the same priorities as its updates. The main consequence is that an update which arrives is no longer necessarily in the same relationship (i.e., higher or lower priority) with all of the ones in E. The revised version of Algorithm 2 that we get is detailed below, from which the reader should be able to deduce the corresponding amendments to Algorithm 1.

The nodes keep the same state: queues Q_i and E_i. They behave as follows:

- If N_i generates an update u, then u is executed locally and u is inserted at the tail of Q_i.
- If Q_i is nonempty and there is a space available on the ring, then the head of Q_i is removed and inserted into the ring, and into the tail of E_i.
- If an update u' returns which was originated by N_i as u (but may have been altered since), then it is removed from the ring, and from the head of E_i.
- If an update u arrives that originated at another N_j, then for each of the entries v in E_i in turn, starting with the head (i) if $u < v$ (i.e., v has higher priority than u) then u is replaced by u^v and (ii) if $v < u$ then v is replaced by v^u. Notice that, since u is changing as this procedure progresses, it is necessary to carry it out in this strict sequential order. Finally, if u' is the version of u that emerges at the end of this process, then u' is passed on round the ring and u'^{Q_i} is executed locally.

One obvious application of this generalised procedure is where the priority of updates is determined by a *timestamp*, in other words an indication of the time when it was inserted into the ring. The later the timestamp, the higher the priority. We would, of course, have to put in some tie-breaking mechanism (perhaps based on node priorities) in the case where it is possible that two nodes can generate updates with the same timestamp.

Suppose first that updates are timestamped as they enter the ring. This is most likely to be the case where there are no Q_i. Provided that the clocks on the nodes are sufficiently close that no update u from node N ever passes node M before M emits an update with a timestamp that precedes u's, then the overall effect of the algorithm is to execute, at every node, a sequence of updates equivalent to the updates, as emitted from their origins, in timestamp order.

It is sufficient to show this for the one update per node case which we dealt with in most of the proof of Theorem 4.1. Observe that, under this assumption and our assumption about the clocks above, the timestamp order is consistent with the partial order $\prec$ in the sense that, given any nonempty subset of the set U of updates, the one with the latest timestamp is $\prec$-maximal. We know (by the proof of Theorem 4.1) that $f(U)$ is equivalent to the set of updates executed at each node. If $U = \{u_1, \ldots, u_n\}$, where $i > j$ means that u_i has the later timestamp, then $f(U) = u_1; \ldots; u_n$. This is easy to prove by induction since u_r is always $\prec$-maximal in

$\{u_1, \ldots, u_r\}$, and $u_r^{\boxed{g(\{u_1,\ldots,u_{r-1}\},u_r)}} = u_r$ since every element of $\{u_1, \ldots, u_{r-1}\}$ has lower priority than u_r.

If all nodes are keeping their own clocks and we do not want the type of inconsistency banned in last paragraph, then an obvious way round it is for each node, upon receiving an update from elsewhere with a later timestamp than the time on its own clock, to advance its own clock accordingly.

If the nodes do keep Q_i and the messages are timestamped on their generation, then the modified algorithm presented above does not keep the right discipline of conjugation between arriving updates and updates in Q_i. Clearly the Q_i will have their earliest updates at the head, and latest at the tail. Some of the updates at the head of a given Q_i might be timestamped earlier than a given u that arrives, and the remainder later. Intuitively we would expect the correct thing to do would be to replace the final sentence of the algorithm above with the following.

- Finally, if u' is the version of u that emerges at the end of this process, then u' is passed on round the ring. If $Q_i = H; T$, where H consists of the elements of Q_i with earlier timestamps than u, and T consists of those with later timestamps, then H is replaced by H^u and u^T is executed locally.

The author conjectures that this algorithm works, in the same sense as above, under the same condition – no update ever arrives at a node, which now means the part of the node generating the updates, with later timestamp than a subsequent one emitted by the node itself. He believes that the proof of this would revolve around showing that the sequence of nodes executed at each node is $f(U)$, defined with respect to a subtly different $\prec$. This would not be the concurrency order defined on just the ring traffic, but on the whole network including the paths between the ring and the parts of nodes which generate updates. We would have $u \prec v$ if and only if (a version of) u is executed at every node before (a version of) v, or equivalently if u is executed at v's origin before v is generated and inserted into the appropriate Q_i. There will be more discussion of this point in Section 6.

5 More about conjugate algebras

In this section we will discuss some more general properties of conjugate algebras, give some more examples of them and describe some techniques for combining them together. It should provide a basis for those interested in devising languages of updates that are allowable for the algorithm presented in the last section.

As well as commutative monoids, we have already seen two nontrivial examples of conjugate algebras: any group under its usual conjugation operator and constant assignments to areas of store. These can be reasonably thought of as extremal: in the one case everything is fully reversible in a very strong sense (the algebra has inverses) and in the second case practically nothing is. In this section we will see others between these extremes. Before going into the theory we will describe one such, which generalises both the assignment example and a simple example of a group.

We assume that the values over which variables range are the real numbers or positive and negative integers, or some other commutative ring with a unit. Instead of just allowing sets of assignments of the form $x := c$, with c constant, we will now take as our model sets of assignments of the form $x := a + bx$. Indeed, it is convenient to assume that there is an assignment in the set for every variable, since the absence of an assignment to a variable x is equivalent to the presence of $x := 0 + 1.x$.[12] The sense in which this generalises a group is that if the value-space is a field (e.g., the reals) and the constant b is always non-zero (i.e., constant assignments are specifically excluded) then the set of updates that arise is a group.

For simplicity in what follows we will assume that there is only one variable, x, but exactly the same constructions work for the general case, with different variables being quite independent. (The general case is just the *product* of simpler ones: see below.) We will denote the update $x := a + bx$ by the ordered pair $\langle a, b\rangle$. The sequential composition of two updates is easily computed:

$$\langle a, b\rangle; \langle c, d\rangle = \langle c + ad, bd\rangle$$

and from this it is reasonably easy to devise a conjugation operator

$$\langle a, b\rangle^{\langle c,d\rangle} = \langle ad + (b - 1)c, b\rangle$$

which satisfies $L2$ (i.e., $\langle a, b\rangle; \langle c, d\rangle = \langle c, d\rangle; \langle a, b\rangle^{\langle c,d\rangle}$). A little computation reveals that this operator also satisfies $L3$ and $L4$. The unit assignment is $\langle 0, 1\rangle$, which trivially satisfies all that is required of it.

Unfortunately this example does not generalise in the obvious way to the case where, instead of the assignment to each variable being linear in its own value, it is now linear in a combination of variables. In the case where there are only finitely many variables, and $\underline{x}$ is a list of them all, one could write such updates in the form $\underline{x} := \underline{c} + A\underline{x}$ for $\underline{c}$ a list of constants and A a square matrix. As an example to illustrate this impossibility, consider the case of two integer variables with

$$e \equiv \langle x, y\rangle := \langle x + y, 0\rangle \quad \text{and} \quad f \equiv \langle x, y\rangle := \langle x, x\rangle$$

The combined effect of $e; f$ is to assign $x + y$ to both variables. But, since f forgets the value of y, there is no update e^f which, composed $f; e^f$, can achieve the same effect for all x and y.

Provided the value space is a field one can get a working algebra over a finite list $\underline{x}$ of variables (and it worth noting that the set of all variables might be partitioned into a number of self-contained classes) where every update is *either* of the form

$$\underline{x} := \underline{c} + A\underline{x}$$

for a nonsingular A, *or* a constant assignment to the whole of $\underline{x}$. The reader might be interested to work out the details. If we think of the list $\underline{x}$ as a unit, this and the earlier example both have the property that every update is either well-behaved in one way or is a straight constant assignment to the object in hand which ignores the earlier state. We will later extend this idea into a general construction: *annihilating sum* for building conjugate algebras.

We can look at the theory of conjugate algebras from two distinct standpoints. One is simply to see what we can derive from the axioms, the other is to look at how they act on sets (which corresponds to examining the functional behaviour of families of updates). In what follows we will examine both of these topics briefly, though it seems likely that there is far more to learn about the structure and actions of the algebras than we have space for here.

[12] To add a touch of credibility to this example, we could imagine that the database consisted of bank balances, and that allowable actions include clearing (zeroing) an account, adding or subtracting a constant for a deposit or withdrawal, or multiplying by a constant to apply interest. A banking system implemented using this algebra would have superior properties to one implemented using precomputed constant assignments, for the reasons discussed at the start of Section 4.

First, some obvious definitions.

DEFINITIONS Given a conjugate algebra G whose identity element is 1, a *subalgebra* is a subset H which contains 1 and which is closed under the operations of sequential composition (;) and conjugation. A *unit* is an element a with an inverse a^{-1} such that $a; a^{-1} = a^{-1}; a = 1$. (If an inverse exists for an element a it is easy to show it is unique: consider $x; a; y$ for inverses x and y.) A *subgroup* is a subalgebra which is a group under ;. (This is the same as a subalgebra all of whose elements are units.)

A *homomorphism* is a map $\phi : G \to H$ from one algebra to another such that $\phi(1_G) = 1_H$ and both operations are preserved, namely

$$\phi(a; b) = \phi(a); \phi(b) \qquad \text{and} \qquad \phi(a^b) = \phi(a)^{\phi(b)}$$

The *kernel* of a homomorphism $\phi : G \to H$ is $\{a \in G \mid \phi(a) = 1_H\}$. It is trivially a subalgebra.

Given an family of conjugate algebras G_λ indexed by a set Λ, we can make make their cartesian product

$$\prod_{\lambda \in \Lambda} G_\lambda$$

into a conjugate algebra by defining

$$(\underline{x}; \underline{y})_\lambda = x_\lambda; y_\lambda \qquad \text{and} \qquad (\underline{x}^{\underline{y}})_\lambda = x_\lambda^{y_\lambda}$$

It is easy to check that this construction works. ∎

In a general monoid, for an element a to have an inverse it is not enough for it to have just a right inverse, namely x such that $a; x = 1$. The extra structure of conjugate algebras does give us this, however, since we have that

$$a = a; a; x = a; x; a^x = a^x$$

and hence

$$1 = a; x = x; a^x = x; a$$

or, in other words, x is also a left inverse for a. It is easy to see that this argument also shows that any element (x) with a left inverse (a) also has it as a right inverse.

It is worth noting that the conjugate of a unit is a unit, since if $a; b = 1$ then $a^x; b^x = 1^x = 1$, and that if a is a unit and x arbitrary then

$$x^a = a^{-1}; a; x^a = a^{-1}; x; a$$

which means that x^a is determined in this case by sequential composition, and corresponds precisely to the definition of conjugation in groups.

Indeed, conjugation by any fixed element is always a homomorphism from a conjugate algebra to itself. This is trivially guaranteed by the axioms. If the element is a unit then it is an automorphism (homomorphism that is a bijection).

There are a number of possible definitions of what it means for a subalgebra N to be *normal*, none of which seems to be totally satisfactory. All are equivalent to the usual definition in the case where the overall algebra is a group. In the following we use the notations X^a, a^X, $a; X$ and $X; a$ for a subset X of G to mean $\{x^a \mid x \in X\}$ etc.

(a) $N^g \subseteq N$ for all $g \in G$.

(b) $g^N \subseteq g; N$ for all $g \in G$.

(c) $N; g \subseteq g; N$ for all $g \in G$.

It might seem a little odd that we have used $\subseteq$ rather than $=$ in cases (a) and (c), for equality trivially holds in the case of a group if the inequality does. They have been phrased in this way though, to ensure that the group of units U of G is always normal: a demand that seems eminently reasonable and which would not always be satisfied with $=$ (see annihilating sums below). It is easy, in general, to prove that (a) implies (c), and that if N is a group then (a) implies (b) and that (b) and (c) are equivalent. The kernel of any homomorphism is easily shown to satisfy (a).

The author has discovered two different ways of forming quotient spaces, which coincide in the case where N is a group, but which do not seem to be especially useful in more general circumstances. The first, in the case of N satisfying (a), is to declare $a \equiv b$ if there are n and m in N such that $a; n = b; m$. The second, which works for N satisfying (a) and (b), is to declare $a \equiv b$ if $a; N = b; N$. In each case the quotient space defined in the obvious way is a conjugate algebra, with the quotient map a homomorphism. (The well-definedness of the first actually depends on the the properties of the algebra, since if $a; n = b; m$ and $b; p = c; q$, then

$$a; n; p^m = b; m; p^m = b; p; m = c; q; m$$

which proves transitivity.) Even in the case where the kernel N of a homomorphism ϕ is a group, we unfortunately do not get the general isomorphism theorem $G/N \cong Im(\phi)$. The author's investigations suggest that there may just not be enough structure in the algebra to give a useful notion of normality and an isomorphism theorem of the type above. Perhaps if one had cancellation laws something more could be done.

One type of example which makes normality constructions difficult is where there are a lot of *right zeroes* in the algebra: elements such that $g; z = z$ for all g. We have seen this type of behaviour in the example at the start of this section, where any assignment of the form $x := a$ is a right zero. (In the case of the area assignment example with a single variable, all non-identity elements are right zeroes.) And it is always easy to introduce such zeroes into an algebra: suppose G is a conjugate algebra and Z is a set (disjoint from G) on which G acts – i.e., there is an operator $z * g$ such that $z * 1 = z$ and $z * (g; h) = (z * g) * h$ for all $z \in Z$, $g, h \in G$. Then we can form a conjugate algebra out of $G \cup Z$ – the *annihilating sum* – in which all elements of Z become right zeroes, as follows. Below, 1 refers to the identity of G, which becomes the identity of the new space.

- if $g, h \in G$, then $g; h$ and g^h are as in G.
- $a; z = z$ for all $a \in G \cup Z$ and $z \in Z$.
- $z; g = z^g = z * g$ for all $g \in G$ and $z \in Z$.
- $a^z = 1$ for all $a \in G \cup Z$ and $z \in Z$.

It is straightforward to verify that this satisfies all of the axioms. Consider, for example, the case of $L3$. If $w \in Z$ then

$$(u; v)^w = u^w; v^w$$

is trivial, since both sides equal 1. If $w \in G$ and both u and v are in G, then the property is inherited from G. So we can assume that $w \in G$ and one or both of u and v are in Z (which in any case imply that $u; v \in Z$). Thus $(u; v)^w = (u; v) * w$. In the cases where $v \in Z$ this equals $v * w$ which in turn equals $(u * v); (v * w)$ as required. If $v \in G$ then $u \in Z$ and so

$$(u; v) * w = u * (v; w) = u * (w; v^w) = u^w; v^w$$

which completes the proof of $L3$.

It is interesting to compare this example with the example at the start of this section when the ring on which it is based has no zero-divisors (i.e., non-zero a and b such that $ab = 0$). This consists of the constant assignments Z (all right zeroes) together with a conjugate algebra G of non-zeroes (assignments of the form $x := a + bx$ for $b \neq 0$). This turns out to be precisely the annihilating sum under the obvious action of G on Z, except that the definition of g^z is different. This is because there is sometimes more than one conjugation operator that will work for a given sequential composition monoid.

It is worth noting that, in the type of examples we are looking at based on database updates, the sequential composition operator is somehow more fundamental in a conjugate algebra than is the conjugation operator. This is in the sense that we are far more likely to ask the question of whether a given monoid can be given a suitable conjugation operator, than the other way round.

We can gain some insight into which sequential composition monoids can be used to form a conjugate algebra by examining their actions. Since we are primarily interested in monoids which are sets of updates on some set S of states, we will assume the monoid is completely determined by its action[13] on a set S, and so can be identified with a sub-monoid of the set of all functions from S to itself (under composition of functions), and contains the identity function on S. If the monoid were a group, then all its functions would be bijections. We are interested in the more general case where this is not the case. If $u \in G$, we will write $x \equiv_u y$ if $u(x) = u(y)$. Suppose $x \equiv_u y$ and v is any other element of G. Then, assuming there is a conjugation operator, we know that there exists w such that $u; w = v; u$, or, in other words

$$w(u(z)) = u(v(z))$$

for all $z \in S$. It follows that $u(v(x)) = u(v(y))$, which can be re-written $v(x) \equiv_u v(y)$. In other words, all the functions in G respect the equivalence relation induced by the others. (It is interesting to note that the failure of this property was the reason why the example discussed earlier using general rather than non-singular matrices failed to work.)

It seems likely that the above result can be taken further, and that similar work can be done to capture just when a monoid of functions can be made into a conjugate algebra. But that will be a topic for further work. For now we will exploit it by constructing a more general and elaborate example of an algebra.

We choose our equivalence relations in advance. Suppose that the state consists of N variables $\langle x_0, \ldots, x_{N-1} \rangle$. We will make life a little simpler if by assuming they all range over the same set X of values, but this is not essential. If $\underline{x}$ is a state we will use $\underline{x} \uparrow n$ to denote $\langle x_n \ldots x_{N-1} \rangle$. The equivalence relations we allow for the updates in our monoid are $\equiv_n$ defined

$$\underline{x} \equiv_n \underline{y} \Leftrightarrow \underline{x} \uparrow n = \underline{y} \uparrow n$$

for $0 \leq n \leq N$. In other words, two states are $\equiv_n$ equivalent if the values of x_n to x_{N-1} are the same. $\equiv_0$ identifies only equal states and $\equiv_n$ identifies them all.

[13] Whether or not the monoid is determined in this way, we can reasonably assume that the monoid acts on the set of states in the sense described when we discussed annihilating sums.

What does it mean for a function u to have $\equiv_u = \equiv_n$? It means that two states $\underline{x}$ and $\underline{y}$ are mapped to the same result if, and only if, $\underline{x} \uparrow n = \underline{y} \uparrow n$. In other words the result must depend only upon the appropriate final segment of the state, and injectively upon that. u must also have the property that it preserves all the other equivalence relations, which mean that the value of $u(\underline{x}) \uparrow m$ depends only on $\underline{x} \uparrow m$ for all m. In other words, the value u assigns to x_m is a function of $\underline{x} \uparrow m$ for all m. Given this fact, it is useful to write u_m as the function which describes u's action on the last $N - m$ components: it maps $\langle x_m, \ldots, x_{N-1}\rangle$ to the values of the corresponding components of all $u(\underline{x})$ when $\underline{x}$ has these last components.

It may be possible to deal with more general cases, but it order to keep the example reasonably simple we will make the additional assumption that, when $\equiv_u = \equiv_n$, the functions u_m for $m \geq n$ are all bijections. In other words, u induces a bijection on the set of states factored by each of the equivalence relations $\equiv_m$ for $m \geq n$. Under this assumption and what we know already, it is possible to describe the possible structure of u completely. For each $m \in \{n, \ldots, N-1\}$ there must be a function g_i from $(N - (m+1))$-tuples to the set of bijections from X to X, and for each $m < n$ there is a function h_i from $(N - n)$-tuples to X such that

- when $m \geq n$, $u(\underline{x})_m = g_m(\underline{x} \uparrow (m+1))(x_m)$, and
- when $m < n$, then $u(\underline{x})_m = h_m(\underline{x} \uparrow n)$.

Any such system of g's and h's describe an allowable u completely.

Let U_n be the set of all u as above, and let U be the union of the U_n as for $0 \leq n \leq N$. The *index* of $u \in U$ will be the unique n such that $u \in U_n$. We will show that U, with functional composition, is a monoid and can be made into a conjugate algebra.

Since the identity map on S is an element of U_0 and functional composition is always associative, to prove U a monoid it is enough to show that it is closed under composition. It is easy to see that the property that $u(\underline{x}) \uparrow n$ depends only on $\underline{x} \uparrow n$ is preserved under composition. Suppose $u \in U_n$ and $v \in U_m$, and that g_i and h_i are functions constructing the components of v as above. If $m \geq n$ then $u; v \in U_m$ since, for each $k \geq m$ $u_k; v_k$ is the composition of two bijections and hence a bijection, and for each $k < m$ then $v(u(\underline{x})) = h_k(u(\underline{x}) \uparrow m) = h_k(u_m(\underline{x} \uparrow m))$ which is of the correct form. If $m < n$ then $u; v \in U_n$: the same argument as above applies when $k \geq n$, and when $m \leq k < n$ then $v(u(\underline{x})) \uparrow k = g_k(u(\underline{x}) \uparrow k+1)(u(\underline{x})_k)$, which is a function of $\underline{x} \uparrow n$ since $u(\underline{x})$ is. Similar considerations apply when $k \leq m$.

If u is an element of U_n we can define a partial inverse for it, u^*, which is an element of U_0, as follows. We can describe u in terms of g_i $(n \leq i < N)$ and h_i $(0 \leq i < n)$. Now construct functions g_i^* for $n \leq i < N$ as follows:

$$g_k^*(\underline{y}) = (g_k(u_{k+1}^{-1}(\underline{y})))^{-1}\,.$$

For $i < N$ define $g_i(\underline{y}) = id_X$ (the identity function). These g_n^* clearly define $u^* \in U_0$, and it should be clear that $u_k^* = (u_k)^{-1}$ for all $k \geq n$. Hence $u^*; u$ and $u; u^*$ are both elements of U_n which leave the last $N - n$ components of the state unaffected.

These u^* can be used to define a conjugation operator in the obvious way: $v^u = u^*; v; u$. Since $v; u$ is only a function of (at most) the components of the state which are inverted successfully by u^*, it follows that $u; u^*; v; u = v; u$, which is what we need to get $L2$. $L3$ holds, or in other words $w^*; u; w; w^*; v; w = w^*; u; v; w$, since the only components of the state which are needed for the final result are certainly contained in those preserved by $w; w^*$. $L4$ corresponds to $(v; w)^*; u; v; w = w^*; v^*; u; v; w$. This holds because $(v; w)^*$ and $w^*; v^*$ both act identically on the components of the state required for computing $u; v; w$: in particular they both act as the inverse of $(v; w)_n$, where n is the greatest index of u, v or w.

Surprisingly, given our experience to date, law $L6$ does not hold (even though $L5$ trivially does with the identity function as identity). This is because, given any $v \notin U_0$, we find $v^*; v \neq 1$. Thus this particular conjugate algebra is not unitary. To make it unitary we can add, as indicated in Section 3, an extra element to act as 1 (the action of 1 on the underlying set is the same as the identity function, but as objects in the algebra they are distinguished). However the existence of a 1, while useful in performing the manipulations which proved our algorithm correct, plays no essential function in the algorithm itself, and adjoining a 1 does not affect the algorithm's operation, and the equality or otherwise of terms from the original algebra is in no way affected by the addition of the 1. This phenomenon of algebras which only satisfy $L1$-$L5$ is worthy of study. It may well prove to be the case that this is the right definition: the main place where $L6$ has been useful to us was in the reduction of all terms of the free algebra to atomic form.

6 Non-ring topologies

Ring topologies have several advantages for our type of system. They are completely symmetric, in the sense that they look the same to all nodes, and when an update returns to its origin the origin knows that all other nodes have seen it. One disadvantage was indicated earlier: the maximum latency, or time which it takes between an update entering the ring and being seen by all nodes, is proportional to the size of the ring. It is also possible that we might be asked to implement one of our systems on a network which is not a ring, and which would be inefficient to build a ring on top of.

In the following subsections we will construct algorithms based on the conjugate algebra model of updates. All of these contain implicitly an algorithm based on the model of updates used in Algorithm 1, with analogues of stopping and cancelling. We leave the interested reader to extract these for himself.

6.1 Joining rings together

One obvious way of creating a more general system is to glue a number of rings together: we can create a special node which sits in two rings and, when it receives an update to execute from one ring, inserts it into the other.

In order to describe these special nodes it is useful to break up the nodes of Algorithm 2 into two parts: a user process U which contains the local copy of the database and generates the updates which originate at the node (and executes them locally before releasing them), and a server process S which holds the queues Q (which sits on its input line from the user) and E, deals with the ring protocol and the necessary conjugations, and sends the correct non-local updates to U for it to execute along an output channel. Except for what is waiting on Q and E, the process S never deals with more than one update at a time. A special node simply consists of two of these S processes back-to-back, one in each of the rings it is connecting. The input line of one is connected to the output channel of the other, and *vice versa*. In setting up the way these two S processes communicate, care will be required to avoid deadlock. For example, at least one of them will have to operate a Q to achieve this.

Given that our ring algorithm works it is easy to see that this special node, joining two otherwise unconnected rings, preserves correctness. For suppose we were to place a copy of the database on the node and execute the updates which emerged from the two output lines in the order they appear. (Since the two S processes can only deal with one update at a time, they cannot swap a pair of updates simultaneously.) So far as each ring is concerned, the sequence of updates executed at the special node is exactly the sequence which would be executed there if the node were part of that ring alone and it generated the updates from the other ring itself. It follows that the sequence of updates at the special node is equivalent to those executed at all the other nodes in the ring and, by symmetry, the other ring.

We remarked above that the special nodes should nly be used to connect a pair of rings at one point. If two rings are joined in more than one place, the result would be that each update would circulate eternally.

Of course, we can use more than one special node to create any tree-connected system of rings. Since there can be many copies of an update circulating at one time in such a system, the latency problem can be much reduced. However the reader may notice that a node can now never tell from the action of the algorithm that its update has finished circulating.

6.2 Tree networks

In the systems described above, we can imagine making the rings smaller and smaller until they just contained one normal node each and enough special nodes to connect them to adjoining rings. If we thought of each of these rings as a single process, joined by the links which connect the two halves of each special node to its neighbours, then we would have a method of constructing an arbitrary tree network. Notice that in this, as in any other way of configuring a tree to send updates generated anywhere all round, updates can pass either way along the links, which was not the case in the ring.

It seems rather odd to be constructing a tree in the way described in the last paragraph, since firstly each node is actually described as a parallel process and secondly one would think that a tree was fundamentally simpler than a ring and ought to have a solution of its own. In fact it is possible to describe an algorithm for operation on an arbitrary tree, though in its actual operation it is rather close (though not identical) to the one set out above.

Conceptually we will build our network out of two types of processes: nodes which hold copies of the database, can generate updates and receive, execute and pass on updates from elsewhere, and processes which sit on the links between the nodes and regulate updates queued up and passing each other. A node will deal with only one update at a time: either it is one it generates itself, in which case the update is executed and passed to the links with all neighbours, or it is one received from a link, in which case the update is executed and passed to all other links. (Note that, in this last case, if there are no other links because our node is a leaf, the update is passed to nowhere else.)

The processes on links each keep two queues moving in opposite directions. No overtaking is allowed in either direction. When a pair of updates u and v pass each other moving in opposite directions, the link process must conjugate one (and only one) of them by the other. So after u and v pass they become either u and v^u or u^v and v. There are a number of strategies the process could use for this: it could choose on some priority basis depending on the identifiers of the updates, it could always conjugate the one moving in a particular direction, or (and this corresponds to what happens in the case of many small rings discussed above) whenever a new update enters from either direction it could be moved past, and conjugated by, all the updates queued in the opposite direction.

In order to prove this works all we have to do is prove that the updates executed at either end of a single link are the same, since it should be obvious that each update visits each node exactly once (perhaps differently conjugated at different places). This is not too hard to establish, the author's proof being obtained by breaking up the link up into a number of pieces not capable of holding more than one update each (in both directions combined). An update can pass from one to the next if the latter is empty, or if both full they can swap and conjugate one of the two updates. The relationship between the sequences of updates which have passed through consecutive pieces is easy to establish, and when both are empty it is easy to show that these sequences are equivalent. Any finite behaviour of the original link can be modelled by one of the latter form provided it is broken into enough pieces.

There is an interesting contrast between the operation of this algorithm an the ring-based ones. For in the tree case an update becomes conjugated when it 'meets' another update (by passing it on a link), while in rings updates never meet each other and are conjugated by the 'expected-back' copies. The existence of a linear priority order which governed conjugation was essential in the case of rings (as is easily demonstrated by examples). In the tree case there is no such need: it is easy to construct examples of three updates, each of which meets the other two and which conjugate each other cyclically (i.e., a conjugates b, b conjugates c and c conjugates a) without destroying the correctness established above. (In the tree algorithm, as in the ring case, it is easy to see that an update will meet another one – or its expected back version in the case of rings – if and only if there are nodes which disagree on their order, and that if so, they meet exactly once.)

6.3 General networks

The best hope for producing an algorithm which will work on a more general structure would seem to be the functions f and g defined in Section 3. These essentially give us a prescription for implementing a system, since they tell us exactly which are the permissible orders for updates to meet and conjugate each other, based on some consistent notion of priority. It seems clear that the partial order with respect to which the functions are defined will always be the 'causality' one, namely $u \prec v$ if u has been seen at v's origin when v was emitted, and that the routing of updates should be sufficiently deterministic that this order coincides with the 'concurrency' order: $u \prec v$ is all nodes see u before v.

Under these conditions the sequence of updates executed at a given node N is consistent with a presentation of f, in the sense that each successive update u must be either greater than, or incomparable with, each that has preceded it at N. What we must ensure is that, when it is executed at N, u has been conjugated in the box algebra to $u^{\boxed{g(k(N,u),u)}}$, where $k(N,u)$ is, as in Section 4, the set of updates to arrive at N before u. This worked out very naturally in the ring case, and also in the tree case if priorities are adhered to in carrying out conjugation. But in cases where streams of updates are, for example, being merged together, it may well require more ingenuity or, at least, a more technical algorithm, to achieve this. We leave this as a subject for future work.

There is a connection between this 'general topology' discussion and our discussion on timestamping at the end of Section 4. For note that we were conjecturing there that the most complex algorithm devised there worked by reference to a function f defined over the concurrency/causality order on a more general structure – in that case a ring with hairs. An obvious second connection is the question of whether it is possible to extend this work on timestamping to more general topologies. The author believes that in any case where one can come up with a working algorithm, if update priorities are based on consistent timestamps, then the algorithm will deliver the desired result of updating all nodes by a sequence of updates equivalent to the original updates in timestamp order. The argument should once more revolve around the function f, as in our earlier discussions of timestamps. This is another topic for future work.

We discussed earlier how our algebraic theory was connected with, and might be applied to, the subject of true concurrency. It is interesting to note that there is another connection with this topic which appears from the two partial orders (causality and concurrency) which we used to order the updates going round a system. For, if we thing of the total transit of one update as a single event which is observed by a node when it reaches it, then these orders become ones frequently used in true concurrency. Recall that our algorithms have all made sure that these two orders coincided. One could view our algorithms as working by making sure that each node calculates, in possibly different ways, a natural algebraic invariant arising out of a 'true concurrency' view of the way the system works. Thus, in some way, one might view the algorithms in this paper as being a practical application of that subject!

7 Prospects

We have discussed elsewhere possibilities for further work on the algebra introduced in this paper and applications outside databases. The last section provided a little insight into what might be required to generalise our methods beyond the classes of network covered. In this section we will briefly discuss a few more topics which may be profitable subjects for further work related to the application and generalisation of our algorithms.

In Section 5 we began to see just what was and was not allowable for a language of updates which is to be made into a conjugate algebra. It would be interesting to take this further, both by investigating theoretical bounds and by producing further examples. In some sense our three major non-group examples – area assignments, $x := a + bx$ and the last one with its hierarchical equivalences – form a sequence where each more-or-less generalises the previous one. Do all examples follow this general style or are there others which look completely different?

It is perhaps worth pointing out that the symmetric algorithm which appeared in our proof when the box algebra replaced the conjugate algebra is perfectly valid in its own right. If we have a language of updates and a box-conjugation operator which satisfy axioms $L1$ and $P1$-$P5$ then the symmetric algorithm may be implemented directly. (Slight modifications are required in the management of the queues Q_i.) Quite different effects can be achieved.

For example, suppose that the model of updates is constant area assignments as discussed at the start of Section 3, but that additionally there is an error state $\perp$, upon which no assignment has any effect, and a special update z which sets the state to $\perp$. Clearly z is a left and right zero of sequential composition. We can define a box-conjugation operator by setting $v^{\boxed{u}} = v$ if u and v do not clash (i.e., have disjoint domains), and $v^{\boxed{u}} = z$ otherwise. $v^{\boxed{z}} = 1$ for all z, and $z^{\boxed{v}} = z$ for all $v \neq z$. It should not be too hard to see that this satisfies all of $P1$-$P5$. An interpretation of this example is that, for some reason, the simultaneous existence of clashing updates is an error, and when it occurs we want all nodes to know the error has occurred. (A system using this algebra might be implemented as a guard against the break-down of some other protocol which was meant to prevent concurrent updates of the same location.)

The obvious question that arises here is: to what other uses can this symmetrical algorithm be put? The algorithms we devised for other network topologies in Section 6 can also be adapted to use the box algebra.

Acknowledgements

I am grateful to Joy Reed and Michael Goldsmith for reawakening my interest in my original database algorithm after it had slumbered for many years. Michael, and more recently Steve Brookes, have provided a useful sounding board for my ideas and have made a number of useful suggestions. Samson Abramsky pointed out the connections between my box algebra and the work of Gene Stark.

References

M A. Mazurkiewicz, *Trace theory*, In 'Advanced Course on Petri Nets', GMD, Bad Honnef, September 1986.

R A.W. Roscoe, *Routing messages through networks: an exercise in deadlock avoidance*, in Muntean *ed.*, Programming of Transputer Based Machines: Proceedings of 7th occam User Group Technical Meeting, (14–16 September 1987, Grenoble, France), IOS B.V., Amsterdam.

S E.W. Stark, *Connections between a concrete and an abstract model of concurrent computation*, in Proceedings of MFPS 89, Springer LNCS 442.