Moving Between Contexts

Alan Dix

School of Computing and Mathematics
University of Huddersfield
Queensgate
Huddersfield, UK, HD1 3DH
email: alan@zeus.hud.ac.uk

Abstract. Any action is performed in a particular context. So what does it mean to do the 'same' thing in a different context? There is no simple answer to this question , it depends on the interpretation of the operation and even then may be ambiguous. This is not a purely theoretical problem, but occurs in practical computational problems. This paper examines this issue looking at three different problems: multi-user undo, distributed update and the simultaneous development of a document in multiple formats. In each case, we find formal rules which any sensible translation must obey. We also see that dynamic pointers, a generic specification and implementation concept defined in previous work, can be used to generate default translation rules which suffice in many circumstances. This is because dynamic pointers can themselves be seen as a translation of location information between different contexts.

1 Introduction

In this paper we are going to look at three different situations where operations which have been formulated in one context have to be reinterpreted in another. If you stand up in the morning, it is pretty clear what it means to do the same thing in the evening. But, if you face the east in the morning, what is the equivalent action in the evening? Do you still face east, or do you face west towards the setting sun? As is evident, a general answer to the question involves the meaning and purpose of an operation. However, we will see that there are many situations where there is an obvious and computable meaning to this concept.

The first situation we will consider is the issue of undo in a multi-user interactive system. One of the solutions to this involves switching the order of commands. So, the first command has to be understood in the context before the second and vice versa.

In the second example, we will consider the simultaneous update of the same object in a distributed environment. In order to merge these updates it is necessary to reinterpret updates as if they had been performed one after another, rather than simultaneously.

Finally, and perhaps most exciting, we will look at the parallel development of related documents in different formats. One of the most difficult things about cooperative writing is finding a single word-processor or text proocessing package to use. Often one resorts to using plain ASCII files! However, we will see that it is possible to work on different formats translating updates formulated in one format into equivalent ones on the other.

Throughout the paper, we will consider the general properties required of different forms of translation between contexts. In addition, we will see how dynamic pointers, which encapsulate the translation of location information between contexts, can be used as a generic mechanism to aid in the production of appropriate translation rules. The properties of dynamic pointers are dealt with in detail elsewhere [3] and their application to multi-user interface issues has also been developed [2]. In this paper we will summarise sufficient of this background to make the paper self-contained.

2 Group and long-term undo

2.1 The problem

Providing undo support has been a challenge for several synchronous multi-user editors [8]. The principal problem is as follows:

Two users, Alison and Brian, are editing a document. First Alison performs action a, then Brian performs action b, finally Alison presses the undo button – what happens? There are two options.

global undo – the very last action b is undone
local undo – Alison's last action a is undone

The meaning of global undo is clear and, although it is often expensive to implement, at least you know what to do. Unfortunately, in all but the most tightly coordinated editors this is not the behaviour a user would expect. Local undo is what they want.

Although the description of this problem is in terms of group undo, it is important to note that the same problem arises in single user systems. If you had done both actions yourself and then realised that the first was wrong you would be in exactly the same situation.

2.2 Commutativity

Gregory Abowd and I examined the general issues using a formal model of group undo [1]. The formal model allowed us to clarify when local undo could be given a sensible meaning. This was when the users' commands commute. If the result of ab is the same as ba, then one can simply pretend the commands happened the other way round and effectively use global undo.

Figure 1 shows this process. You can think of this as rewriting history! It is rather similar to the serialisation conditions familiar in database theory.

In fact, it is not necessary that the operations commute in all contexts, but merely that when applied to the particular state s_0 where we started. That is, if $doit$ is the state update function, we require that:

$$doit(a, doit(b, s_0)) = doit(b, doit(a, s_0))$$

It is certainly safe, if we can sow that commands commute in all circumstances, but this is conservative. We might find that some pairs of commands commute only sometimes and if we can detect these we can undo them in the circumstances where they do commute.

Unfortunately, not all comands commute, so we also considered various ways of making commands more likely to commute. These include locking (which prevents people performing non-commuting actions), and structuring the document into sub-objects so that updates to

$$\begin{tikzcd}
 & S_a \arrow[dr, "b"] & & \\
S_0 \arrow[ur, "a"] \arrow[ddr, "b"'] & & S_{ab} \arrow[d, phantom, "\|"] & \\
 & & S_{ba} \arrow[r, "\text{undo}"] & S_b \\
 & S_b \arrow[ur, "a"'] & &
\end{tikzcd}$$

Fig. 1. Local undo = rewrite history + global undo

different objects can commute. The hardest case is free text. This is because an update in the text affects the offsets of characters in the whole of the rest of the document. No implemented group undo system of which I am aware caters for multiple actions to the same text object.

Happily, there are ways of representing operations on text so that they are more likely to commute. This was explored in [1] when we suggested the use of a variant of Ellis and Gibbs distributed update algorithms and also in [2] where the dynamic pointer solutions were explored in detail. In both methods the operations are translated when they are reversed. For example, imagine the operations were as follows (a sort of mutual admiration society):

s_0 = “`Alison␣is␣␣and␣Brian␣is␣.`”
a = insert(24,“`beautiful`”)
b = insert(10,“`adorable`”)
s_{ab} = “`Alison␣is␣adorable␣and␣Brian␣is␣beautiful.`”

If we reverse these operations the effect is disasterous.

s_{ba} = “`Alison␣is␣adorable␣and␣Bbeautifulrian␣is␣.`”

Both the use of Ellis and Gibbs algorithm and dynamic pointers effectively modify the first operation slightly to give:

b = insert(10,“`adorable`”)
a' = insert(32,“`beautiful`”)
$s_{ba'}$ = “`Alison␣is␣adorable␣and␣Brian␣is␣beautiful.`”

Just what we wanted!

2.3 Levels of interpretation

There is something very strange happening here. Surely operations either commute or they don’t? How can you change the representation of something and make it commute when it didn’t before? Let’s go back. We said that undo is possible if two operations a and b commute, that is if:

$$doit(a, doit(b, s_0)) = doit(b, doit(a, s_0))$$

However, in the example we gave, the operations on either side were not the same – we rewrote one of them to give a'. We cheated! The original operations didn't commute, we just invented some different operations which had the same effect. This sounds like very shaky ground on which to stand.

In fact, the problem is more fundamental. It is not clear that the original concept of commuting updates is well formulated. If you asked anyone what it would mean to do the two insertions a and b in the opposite order they would give the dynamic pointers answer, not the one based on the literal operation. Of course, if we had instead described the operations as:

a = insert(at Alison's cursor, "`beautiful`")
b = insert(at Brian's cursor, "`adorable`")

Then we would have expected that Alison's cursor would be in a different position before and after Brian's operation. (In fact, this is effectively how dynamic pointers work.) In any system there will be multiple interpretations of the same user action at different levels in the system. At the physical level we will have interpretations like 'click mouse at 117, 523', a little deeper, this would be interpreted as 'click the "delete" button', this would then translate to the internal operation 'delete the text at the selection', which finally might become 'remove characters 573 to 597 from the text'. Of course, in the user's head this might just be 'oops, didn't mean to hit the paste key'. So, the same operation may be described in different ways and, depending on which we choose, the 'same' operations may or may not commute.

Arguably the best interpretation will be the one the user means. However, there is evidence that the user's idea of what is being undone differs from that of the system designer, certainly at the level of the grouping of actions for undo [10]. Also, not all users have the same interpretation and so there is no gold standard.

2.4 Changing contexts

How come these different levels of interpretation differ so much in their meaning when we consider undo? Are systems poorly managing the translation between user's intentions and the realisation within the machine? While this may be the case, it is not the fundamental problem here. In the above description of different levels, each interpretation meant the same thing – in the context in which they acted. But, if we want the 'same' operation in a different context, the problems start. At each level of interpretation, we can see changes in context which invalidate the description.

(i)	click mouse at 117, 523	—	the window has been moved
(ii)	click the "delete" button	—	the button has changed its function
(iii)	delete the text at the selection	—	the user has selected a different piece of text
(iv)	remove characters 573 to 597	—	text has been inserted or deleted before position 573

So, whatever level of interpretation we pick we need to translate the operation as we move it beween contexts. So, our original formulation of commutativity was flawed, it is normally meaningless to talk of:

$$a\,b = b\,a$$

The second operation b was formulated in the context *after* a was performed (s_a) and so cannot simply be performed in the context before. Similarly, a was formulated in the original context s_0 and so also needs transformation. The result is more like:

$$a\,b = b'\,a'$$

However, the translation that changes a to a' is different from that which changes b to b', so we introduce specific notation for the two translations:

$\overleftarrow{b}^{a}$	—	b moved back to execute before a
$\overrightarrow{a}^{b}$	—	a moved forward to execute after the translated b

These translations must obey the commutativity law making the diagram in figure 2 commute. These translations may also depend on the original context, but in all the examples we will deal with they turn out to be independent of it. Note also that translations will in general be partial – there will be some pairs of operations for which there is no sensible reversal. This happens when commands interfere with one another and we will see an example of this below.

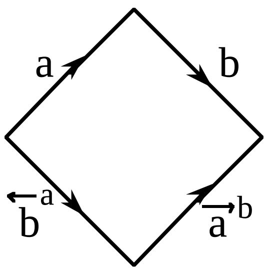


Fig. 2. Undo translations commute

Given this complexity, surely one should design systems so that commands do not depend on context? The design of relational databases means that updates are usually expressed in an apparently context independent fashion. Rather than say 'update that record', you say 'change the salary of employee no. 573 to £17,000'. First, this sort of descriptive formulation is hardly what one would expect of a modern interactive system! Also the context independence is partly illusory. If a previous update had been 'change the employee no. of 235 to 573', then we would have had similar problems to positional commands.

2.5 Example translations

Independent objects. The very simplest translation scheme is not to allow any reordering, that is, the translation is not just partial, it is empty! One step better are translation schemes which divide the document[1] into separate objects. Updates to each object commute with those for other objects and no translation is necessary. If two updates refer to the same object

[1] The word document here refers to the whole thing being updated, whether it is a database, spreadsheet or text

then they are deemed incompatible and no translation is given. Of course, the objects may themselves be split into independent sub-objects or attributes and then updates to different attributes of the same object would be allowed.

For example, in a drawing package each shape on the screen is a different object. Each shape has various attributes: colour, position, width, height. These attributes are changed by direct manipulation and turned into internal operations:

(i)	select blue in colour menu	—	`set_attr(#179,colour,blue)`
(ii)	drag circle across screen	—	`set_attr(#63,position,(356,813))`
(iii)	press delete key	—	`delete(#432)`
(iv)	select box in new menu	—	`create(#112,box)`

The numbers in each operation refer to the internal reference for the relevant objects. In the cases of (i) and (iii), these would be of the currently selected object and in the case of (ii) it would be whatever object was dragged. The final example (iv) is different in that the reference would not be part of the original command, but would be generated when the object was created. However, it would need to be remembered in any history which is used for undo purposes. The translation alters no operations, but some translations are not allowed (the translation is the identity over a restricted domain). The forbidden combinations are:

$\texttt{create}(n, shape)$	&	$\texttt{set\_attr}(n', attr, val)$	—	if $n = n'$
$\texttt{create}(n, shape)$	&	$\texttt{delete}(n')$	—	if $n = n'$
$\texttt{set\_attr}(n, a, v)$	&	$\texttt{set\_attr}(n', a', v')$	—	if $n = n'$ and $a = a'$ and $v \neq v'$
$\texttt{set\_attr}(n, a, v)$	&	$\texttt{delete}(n')$	—	if $n = n'$

If we assume that the program never reuses reference numbers, then combinations such as $\texttt{delete}(n)$ followed by $\texttt{create}(n, shape)$, could never occur and so all `delete`–`create` and `delete`–`set_attr` are compatible.

Text editing. As we noted earlier, independent objects are easy. It is text which causes problems. We'll consider only single character insertions and deletions. The two operations are:

$\texttt{insert}(n, c)$ — insert the character c just after the nth character.

$\texttt{delete}(m)$ — delete the mth character.

First some of the simple rules:

(i) $a = \texttt{insert}(n, c)$ $\quad b = \texttt{insert}(m, c')$ $\quad — n < m$

$\overleftarrow{\mathrm{b}}^{\mathrm{a}} = \texttt{insert}(m + 1, c')$ $\quad \overrightarrow{\mathrm{a}}^{\mathrm{b}} = \texttt{insert}(n, c)$

(ii) $a = \texttt{insert}(n, c)$ $\quad b = \texttt{insert}(m, c')$ $\quad — n > m$

$\overleftarrow{\mathrm{b}}^{\mathrm{a}} = \texttt{insert}(m, c')$ $\quad \overrightarrow{\mathrm{a}}^{\mathrm{b}} = \texttt{insert}(n + 1, c)$

(iii) $a = \texttt{insert}(n, c)$ $\quad b = \texttt{delete}(m)$ $\quad — n > m$

$\overleftarrow{\mathrm{b}}^{\mathrm{a}} = \texttt{delete}(m)$ $\quad \overrightarrow{\mathrm{a}}^{\mathrm{b}} = \texttt{insert}(n - 1, c)$

The nasty examples are where insertions and deletions happen at exactly the same place. Some of these obviously do not admit a sensible reversal, for example, an insert followed immediately by a deletion at the same location. The only reasonable translation is for both to become no-ops.

(iv) $a = \texttt{insert}(n, c)$ $\quad b = \texttt{delete}(n + 1)$

$\overleftarrow{\mathrm{b}}^{\mathrm{a}} = \epsilon$ $\quad \overrightarrow{\mathrm{a}}^{\mathrm{b}} = \epsilon$

However, it would seem better to simply regard these as undefined.

The other combinations have sensible translations, for example:

(v) a = `insert`(n, c) b = `insert`(n, c')
$\overleftarrow{b}^{a}$ = `insert`(n, c') $\overrightarrow{a}^{b}$ = `insert`(n, c)

(vi) a = `insert`(n, c) b = `insert`$(n + 1, c')$
$\overleftarrow{b}^{a}$ = `insert`(n, c') $\overrightarrow{a}^{b}$ = `insert`(n, c)

(vii) a = `delete`(n) b = `delete`$(n - 1)$
$\overleftarrow{b}^{a}$ = `delete`$(n - 1)$ $\overrightarrow{a}^{b}$ = `delete`$(n - 1)$

It is easy to verify that all these rules obey the commutativity property.

3 Merging updates

A similar problem arises when several users are editing an object on a distributed platform. If network speeds allow, the updates can be routed through a central server. In this case there is one document and the only problems which arise are the inevitable race conditions (e.g., two people attempt to type at the same position and get their typing mixed up). However, often the loss of interactive performance for a centralised architecture makes it impractical. Instead, each user's copy of the application will hold a local copy of the shared document. In these cases, one must either lock the objects so that only one person can edit at once, or else accept the risk that two people will edit the same object. If this happens, then the local copies of the documents on the two user's machines will be in conflict, neither will be the 'most up to date' and if one is taken rather than the other, one user's updates will be lost. The problem is how to perform updates on each copy in order to resynchronise the two copies, but in such a way that the two updates are merged, rather than one lost.

In the Grove editor, Ellis and Gibbs designed an algorithm to recover from this situation [5]. Imagine that one user has performed an operation a and the other has perfomed b. Their algorithm finds operations a' and b' such that the diagram in figure 3 commutes.

$$\begin{tikzcd}
 & S_a \arrow[dr, "b'"] & \\
 & & S_{ab'} \arrow[dd, equal] \\
S_0 \arrow[uur, "a"] \arrow[ddr, "b"'] & & \\
 & & S_{ba'} \\
 & S_b \arrow[ur, "a'"'] &
\end{tikzcd}$$

Fig. 3. Ellis and Gibbs' translations

They define translation rules similar to those we have seen for undo. This depends on the order in which the operations ‘should’ have happened. We will assume that a is the first. For example, the rule for insert–insert is:

$$a = \texttt{insert}(n, c) \qquad\qquad b = \texttt{insert}(m, c')$$

$$b' = \begin{cases} \text{insert}(m, c') & \text{if } n > m \\ \text{insert}(m, c') & \text{if } n = m \\ \text{insert}(m+1, c') & \text{if } n < m \end{cases} \qquad a' = \begin{cases} \text{insert}(n, c) & \text{if } n > m \\ \text{insert}(n+1, c) & \text{if } n = m \\ \text{insert}(n+1, c) & \text{if } n < m \end{cases}$$

Notice that the rules for a' and b' differ. This is because if the two users insert at the same location the result depends on which we regard as happening first. Basically, we have to decide whether the final text has cc' or $c'c$. The rules above produce the latter result. Rules for the former would be the other way round, but still a' and b' would differ.

3.1 Moving between contexts

Again we have the problem of translating operations formulated in one context into a different one. This time the commands a and b are formulated in the context of the original state, s_0. We wish to translate a into the context of s_b and b into that of s_a, remembering that a should happen ‘first’. As these translations are similar to those for the undo case, we will introduce similar notation.

$\underset{\longleftarrow \mathrm{a}}{\mathrm{b}}$ — b, which should have happened after a, moved forward to execute after a

$\underset{\longrightarrow \mathrm{b}}{\mathrm{a}}$ — a, which should have happened before b, moved forward to execute after b

Any translation for this sort of context change should obey the following law, and hence make the diagram in figure 4 commute.

$$a \underset{\longleftarrow \mathrm{a}}{\mathrm{b}} = b \underset{\longrightarrow \mathrm{b}}{\mathrm{a}}$$

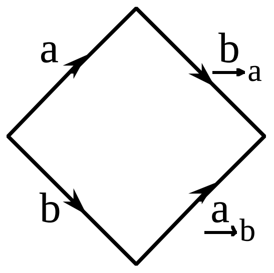


Fig. 4. Moving between contexts for simultaneous actions

As with the translations for undo, it is not clear that these translations can be total. In general we would expect to find some conflict. Ellis and Gibbs managed to develop a complete set for single character operations, but more complex block oriented operations are bound to lead to incompatible updates. For example, suppose Alison performed a global substitution of ‘transformation’ to ‘change’ and Brian simultaneously substituted ‘translation’ to ‘transformation’. In this case, one can think of translations which obey the relevant properties, but it is not at all clear that they are sensible.

In a synchronous editor, it would be very disconcerting to have some of your updates suddenly disregarded, but very intrusive for the system to request clarification on problematic merges. In such systems it is probably best to employ locking for potentially conflicting updates, but only where translations cannot be found. Note that this means analysing in advance those circumstances under which conflicts can arise – it’s no good asking for the lock *after* the user has done the operation!

3.2 Long term interaction

A similar problem arises in more long term interactions. For example, two users have portable machines which are not connected to a network. When they meet and connect their machines any inconsistencies between their data must be resolved. Systems which support this sort of activity, for example Lotus Notes, Laplink or the CODA distributed filesystem [7], work by keeping one copy or other of the data. At best, they detect that an inconsistency has occurred and either warn the user or, in the case of Lotus Notes, keep both copies. One way to tackle the problem is to use translation techniques similar to those above. However, in the case of long term interactions, it is likely that there will be many incompatible updates. This is acceptable as the process of merging can afford to be more heavy weight than in the synchronous case. It is important here that the merging process is made as easy for the user as possible using translation to suggest the most likely form of the merged version. One possibility is to use version managment techniques to allow the users to maintain the different versions of the document [4]. This enables the users to see the history of changes and also gives the system suitable information to present merging in a helpful manner.

3.3 They are all different

The similarity between the different translations suggests that it might be possible to get away with less than four different translations. That is having constructed, say, the two undo translations, one might be able to generate one or other of the merge translations.

If we examine the two commuting diagrams (figure 5), we can focus on different arcs and obtain the following putative identities.

(i) $\overleftarrow{(\underrightarrow{\mathrm{b}}_{\mathrm{a}})}^{\mathrm{a}} = b$

(ii) $\underrightarrow{(\overleftarrow{\mathrm{b}}^{\mathrm{a}})}_{\mathrm{a}} = b$

(iii) $\overrightarrow{\mathrm{a}}^{(\underrightarrow{\mathrm{b}}_{\mathrm{a}})} = \underleftarrow{\mathrm{a}}_{\mathrm{b}}$

(iv) $\underleftarrow{\mathrm{a}}_{(\overleftarrow{\mathrm{b}}^{\mathrm{a}})} = \overrightarrow{\mathrm{a}}^{\mathrm{b}}$

These all sound reasonable, but unfortunately *none of them* hold. The problem is that the translations lose information:

– The undo translations lose information when a is an insert, because in the context before a, the difference between the locations either side of the inserted text is lost.

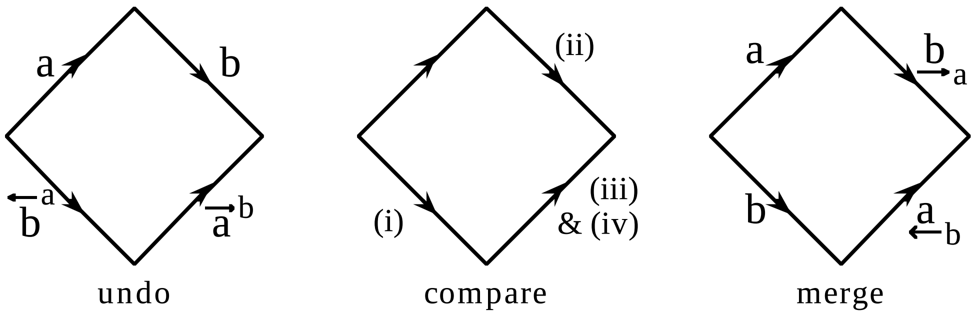


Fig. 5. Comparing translations

– The merge translations lose information when a is a delete, because in the context after a the difference between locations either side of the deleted text is lost.

The problem cases are shown in figures 6 and 7. Each figure shows two commuting diagrams. In figure 6, we see that in an insert–insert scenario you cannot predict the right hand side of the diagram from the left, whilst figure 7 shows that in a delete–insert scenario you cannot predict the bottom from the top.

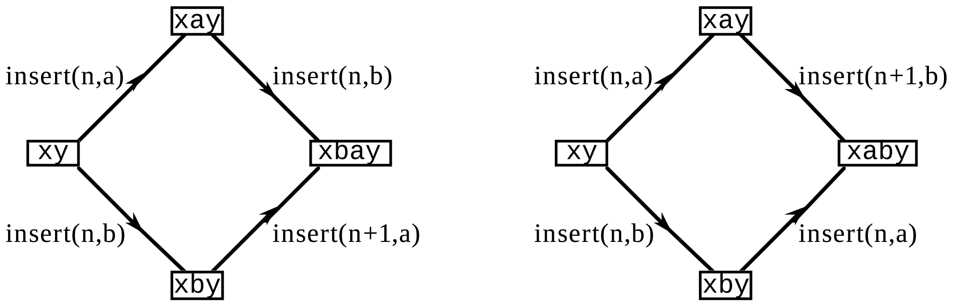


Fig. 6. Problem: the insert–insert scenario

In the insert-insert secenario, identities (i) and (iii) hold. This is because they depend on *first* translating b forward. This does not lose information. However, in (ii) and (iv), b is first translated backwards, which loses information, and hence both fail. In the delete-delete scenario the opposite is true and it is (i) and (iii) which fail. The full counter-examples are given in appendix 1.

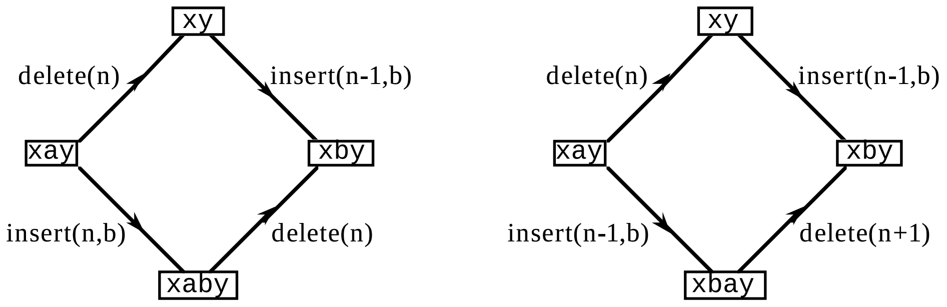


Fig. 7. Problem: the delete–insert scenario

4 Composing translations

So far, we have only seen examples of translations applied to single operations. In fact, for both undo and merge this is sufficient. If you have produced *atomic* translations for each pair of individual commands, then it is possible to construct the composite translations for sequences of commands. These composite translations can easily be built by diagram chasing.

4.1 Undo composition

Consider figure 8. We are trying to find the operations a', b', c' such that:

$$a\,b\,c = b'\,c'\,a'$$

By simply chasing the atomic translations through the commuting diagram we obtain:

(i) $\overrightarrow{\mathrm{a}}^{(\mathrm{bc})} = \overrightarrow{(\overrightarrow{\mathrm{a}}^{\mathrm{b}})}^{\mathrm{c}}$

(ii) $\overleftarrow{\mathrm{bc}}^{\mathrm{a}} \;=\; \overleftarrow{\mathrm{b}}^{\mathrm{a}}\,\overleftarrow{\mathrm{c}}^{(\overrightarrow{\mathrm{a}}^{\mathrm{b}})}$

Similarly, by chasing the commuting diagram in figure 9 we get:

(iii) $\overrightarrow{\mathrm{ab}}^{\mathrm{c}} \;=\; \overrightarrow{\mathrm{a}}^{(\overleftarrow{\mathrm{c}}^{\mathrm{b}})}\,\overrightarrow{\mathrm{b}}^{\mathrm{c}}$

(iv) $\overleftarrow{\mathrm{c}}^{(\mathrm{ab})} = \overleftarrow{(\overleftarrow{\mathrm{c}}^{\mathrm{b}})}^{\mathrm{a}}$

Repeated application of these formulae allow us to calculate the appropriate translation for arbitrary large sequences of commands. Furthermore, they are not just a formal tool, but can be evaluated at run time – an algorithm as well as a definition. Regard each formulae as a rewrite rule left to right and take any expression involving translations and sequences. If we apply the rewrite rules, then we will eventually move the sequence terms to the outermost level and end up with an expression which is a sequence of atomic translations. Each atomic translation can be performed using the given rules and then the elements of the sequence perfomed in turn.

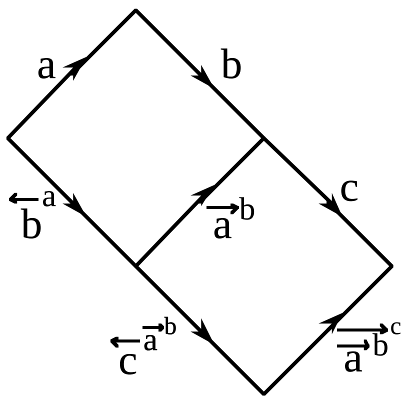

a
b
$\overleftarrow{b}^{a}$
$\overrightarrow{a}^{b}$
c
$\overleftarrow{c}^{\overrightarrow{a}^{b}}$
$\overrightarrow{a}^{\overrightarrow{b}^{c}}$

Fig. 8. Composing undo translations – a(bc)

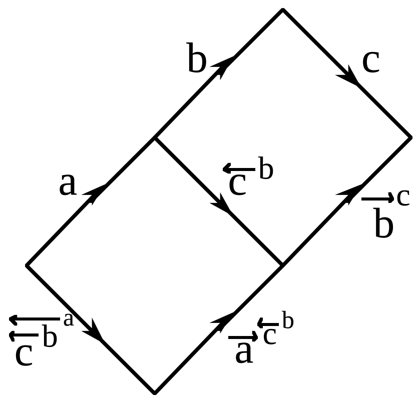

b
c
a
$\overleftarrow{c}^{b}$
$\overrightarrow{b}^{c}$
$\overleftarrow{\overleftarrow{c}^{b}}^{a}$
$\overrightarrow{a}^{\overleftarrow{c}^{b}}$

Fig. 9. Composing undo translations – (ab)c

It is important that these formulae are sound. For example, take an expression like:

$$\overrightarrow{ab}^{cd}$$

We expect to get the same answer no matter what order we reduce expression it. This is pretty clear from the mode in which they were obtained, but a proof that these rules are indeed confluent is found in appendix 2.

4.2 Merge composition

Composition of merge translations follows in a similar manner. The relevant formulae are:

(i) $\underset{\longleftarrow}{a}_{(bc)} = \underset{\longleftarrow}{\left(\underset{\longleftarrow}{a}_{b}\right)}_{c}$

(ii) $\underset{\longrightarrow}{bc}_{a} = \underset{\longrightarrow}{b}_{a}\,\underset{\longrightarrow}{c}_{\left(\underset{\longleftarrow}{a}_{b}\right)}$

(iii) $\underset{\longleftarrow}{ab}_{c} = \underset{\longleftarrow}{a}_{c}\,\underset{\longleftarrow}{b}_{\left(\underset{\longrightarrow}{c}_{b}\right)}$

(iv) $\underset{\longrightarrow}{c}_{(ab)} = \underset{\longrightarrow}{\left(\underset{\longrightarrow}{c}_{a}\right)}_{b}$

Notice that the composition formulae for backwards and forwards translation are duals of one another. This reflects the symmetry of the commuting diagram (figure 4). This means that proofs need tackle only one case, instead of the two needed for undo. Furthermore, (i) and (ii) are identical (swopping appropriate translations) with the cases (i) and (ii) for undo, but I can see no immediate use of this fact.

5 Dynamic pointers

Developing translations for every new operation is obviously a pain. Most of the translations follow a simple pattern. The translated operation is the same as the original, but with the location information modified slightly.

Dynamic pointers are a technique originally developed for coping with single-user interface design [3] and have recently been applied to some of the multi-user problems dealt with in this paper [1, 2]. They encapsulate the way locational information is modified after operations. That is, they capture the translation of pointers between contexts. With any object there is an associated set of pointers wich refer to locations within that object. For example, in the case of text these might be the positions of the gaps between the characters: pointer n refers to the gap between the nth and n + 1th characters and pointer 0 is before the first character in the text. As well as pointers to individual locations within the object, there may be more complex pointers, for example, in pointers to blocks of text or pointers to sets of records in a database.

5.1 Updates – the pull function

For every operation a corresponding ‘pull’ function is defined. This says how the pointers should be updated. For any operation op we will write $pull_{op}$ or simply $pull$ if the operation is obvious. Similarly for any set of objects Obj we write Pt_{Obj} for the set of pointers for those objects or Pt for short. The signatures of the corresponding functions are then:

$op : Params \times Obj \to Obj$

$pull_{op} : Params \times Obj \to (Pt \to Pt)$

The operation will involve some extra parameters (e.g., the text to be inserted) and these have been written generically as $Params$. Note that these parameters will often themselves involve positional information, that is, pointers. This is important later. The pull function needs the same parameters as the operations itself as the partciular form of the update depends on both the parameters (e.g., where the update happens) and the state of the object (e.g., for a delete word operation how big the word is). However, the operation and its parameters will often be obvious from context and so we will usually treat the pull function as a mapping from pointers to pointers.

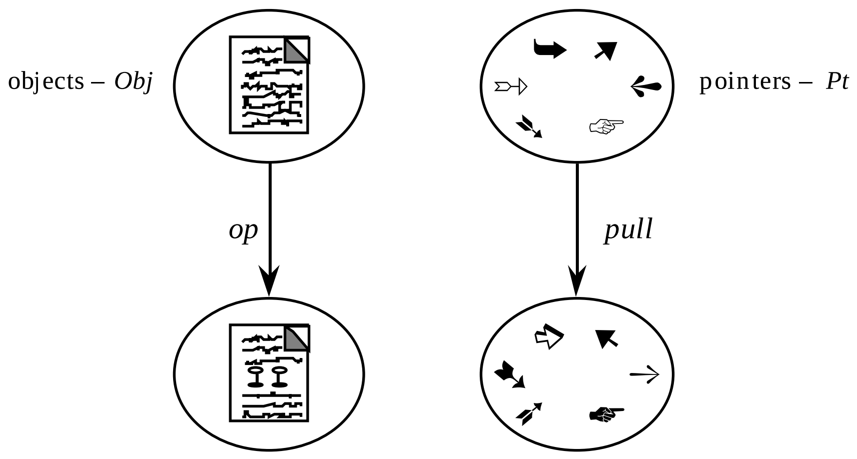


Fig. 10. Dynamic pointers – the pull function

Pull functions for the two character operations are as follows.

$$insert(n, c) \quad — \quad pull_{insert}(p) = \begin{cases} p & \text{if } p \leq n \\ p + 1 & \text{if } p > n \end{cases}$$

$$delete(n, c) \quad — \quad pull_{delete}(p) = \begin{cases} p & \text{if } p < n \\ p - 1 & \text{if } p \geq n \end{cases}$$

Note that it is not clear what the insertion's pull function should do at the point of the insertion. The choice here (the 'leave 'em behind' strategy) is one option, the other being to increment the pointer. In [2] these are called $pull_-$ and $pull_+$ respectively. In older work, the latter choice was taken as the default, but it became apparent from the multi-user situations that this was not the best behaviour and hence more recent work has used $pull_-$ as the primary pull function. A similar problem arises with $pull_{delete}^{-1}$ and this comes in '$-$ and '+' versions. This ambiguity is directly related to the failure of the putative identities in section 3.3.

5.2 Using dynamic pointers for translations

It is evident that the rules for dynamic pointer update closely resemble those required for operation translation. This is reasonable, dynamic pointers transform locations between contexts and the operations are defined in terms of locations. Given an operation *op* defined in terms of pointers from Pt and a pull function $pull$, we can translate the operation to $pull(op)$ by transforming each pointer in the description of op. We can use this to generate default rules for the various translation operations:

$\overleftarrow{\mathrm{b}}^{\mathrm{a}} = pull_a^{-1}(b)$

$\overrightarrow{\mathrm{a}}^{\mathrm{b}} = pull_{\overleftarrow{\mathrm{b}}^{\mathrm{a}}}(a)$

$\underrightarrow{\mathrm{b}}_{\mathrm{a}} = pull_a(b)$

$\underleftarrow{\mathrm{a}}_{\mathrm{b}} = pull_b(a)$

These default rules work everywhere except the immediate area of the change. For the special cases you either have to disallow translations or work out the precise rules by hand. Some of these can be filled in by simply choosing whether you want the $-$ or + versions of the pull functions, but each case has to be checked as this behaviour may not be exactly as required. For undo, it is probably best to signal the conflict rather than guessing, especially if the system also offers more extensive support for the user's own undoing of actions [1]. The synchronous distributed editing situation does demand that all updates are performed silently – the aim is to make the users feel as if they are interacting with a single document. As Ellis and Gibbs show, the rules can be filled in completely for single character edits [5], but for more complex operations this can not be guaranteed. In the case of more long term distributed work, the effort of re-merging can be made explicit and so partial translations are more acceptable.

5.3 More properties of dynamic pointers

Projections. Updates take one object to another of the same type. In addition, we often need to deal with two different kinds of objects with a relationship between them. For example, in an interactive system, we may want to deal with the relationship between a document and its image on the screen. In fact, this is very similar to the update operations. We'll call the two object types Obj and Obj' and the relationship $proj$. There will be corresponding functions relating the two sets of pointers Pt and Pt'.

$proj$: $Params \rightarrow (Obj \leftrightarrow Obj')$
fwd: $Params \times Obj \times Obj' \rightarrow (Pt \rightarrow Pt')$
$back$: $Params \times Obj \times Obj' \rightarrow (Pt' \rightarrow Pt)$

As with the pull function, we will normally drop the initial parameters to the fwd and $back$ functions as these will be obvious from context. The two functions fwd and $back$ translate the pointers in each direction (like $pull$ and $pull^{-1}$) and are weak inverses of one another (i.e., they are inverses on their respective ranges). The two notions could be unified completely, but for compatibility with older work the separate notation is retained here.

Locality of change and sub-object projections. As well as a pull function, each operation can have an associated block pointer representing the extent of the changes in the operation. If this locality of change is supplied for each operation, then it can be used to detect possible incompatibilities. If the locality of change of two operations do not intersect then they can be safely transformed using the default pull translations. The locality of change information can be used statically to determine which combinations require special treatment, or at run time, actually returned as part of the operation allowing detection of incompatibility.

Another extension primarily used during analysis are sub-object projections. These are special projections where one object is apart of another. Sub-object projections can be used to determine the 'normal' behaviour of translations. The definitions we gave allow the possibility of silly translations. For example, for any pair of operations a and b, we can define the translations:

$\overleftarrow{\mathrm{b}}^{\mathrm{a}} = a\,b$
$\overrightarrow{\mathrm{a}}^{\mathrm{b}} = \epsilon$

These obey the commutativity law and are otherwise acceptable, except they are clearly not sensible.

Let L_a and L_b be the locality of change for a and b. Let s_a be any sub-object block pointer which does not intersect L_a and s_b be any which does not intersect $pull_a^{-1}(L_b)$. We write $proj_s$ for the projection which extracts the sub-object at the block s. Then, we expect that:

$$\begin{aligned} proj_{s_a} \circ \overleftarrow{\mathrm{b}}^{\mathrm{a}} &= proj_{pull_a(s_a)} \circ b \\ proj_{pull_{\overleftarrow{\mathrm{b}}^{\mathrm{a}}}(s_b)} \circ \overrightarrow{\mathrm{a}}^{\mathrm{b}} &= proj_{s_b} \circ a \end{aligned}$$

This basically says that $\overleftarrow{\mathrm{b}}^{\mathrm{a}}$ behaves ‘the same’ as b except within the locality of change of a and vice versa.

The full details of the locality of change and sub-object projections are found in [3].

6 Parallel development

6.1 Working together

One of the most difficult decisions in collaborative writing is choosing a common electronic format. Everyone has a favourite wordprocessor or text-processing program, and each package has its own advantages. Not only do these differ in the way they store information, but more important, they differ in the sort of information they store.

For example, let’s consider RTF, SGML and LaTeX. An RTF document (as produced by Microsoft Word) is largely a definition of how a document *looks* – although styles are given names, these are largely an encapsulation of appearance. In contrast, an SGML or HTML document records the structure only, presentation is left to the browser or editor. Somewhere between is LaTeX, popular among computing academics, which has some structure and some layout, although it is certainly not a superset of the other two.

People who are cooperating will want to use their own system when working on the text, but will need to transfer drafts to and from each other. Often this happens at the level of ASCII text (often sent by email). This is a least common denominator representation, throwing away all the advantages of each package. Even with ASCII text there are numerous character set problems due to the different platforms used. Note that we can see this as another form of moving between contexts, this time the contexts are particular software ‘worlds’: the RTF world, LaTeX world etc.

6.2 Translators

A more sophisticated alternative is to use translators between formats. For example, there are converters for RTF to LaTeX, RTF to HTML and LaTeX to SGML. Within a particular platform (PC or Mac) individual wordprocessors will read in many different file formats and there are usually a host of tools to do other translations. If one has a complete set of converters (figure 11) then anyone can work in their preferred format in the knowledge that, if they want to pass it on to another member of the team, they can simply put it through the relevant converter. Unfortunately, as each representation has its own model of text, information is lost

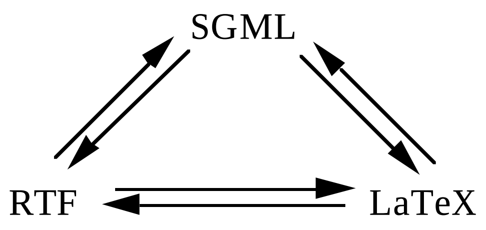


Fig. 11. Individual translators

during the translation process. Depending on the scenario of use this can be mildly annoying or disastrous.

Imagine first that Alison, Brian and Clarise are working on a book together. Alison works in formal methods and is using LaTeX, Brian is in a psychology department and is using MS Word (RTF), and Clarise is working on an Esprit project which is committed to using international standard products and so is using SGML.

In the first scenario each is responsible for particular chapters. When they want comments on a chapter they put it through the relevant converters and send copies to each of their colleagues. As information is lost their colleagues see a slightly different version than the chapter's author. For example, when Brian sends Alison a copy of his chapter, the voice annotations are lost from the Word document, and when Alison sends Brian a copy of hers some of the cross referencing and optional page breaks disappear. The lack of a common appearance and loss of certain features is annoying, but is better than exchanging ASCII.

In our second scenario, the authorship of chapters is shared. One chapter is being worked on by Brian and Clarise. After Brian has worked on a chapter for a while he passes it on to Clarise. Later Clarise will pass the amended version back to Alison. To avoid dealing with merging issues we assume that they make sure that only one person is working on a chapter at a time! When the RTF document is converted to SGML, much of the presentation formatting information is lost. If the translator is quite clever it will parse the style names intelligently in order to produce good markup, but much is bound to be lost. When the reverse translation is done, more will be lost, any mark-up that Clarise added will be lost. Even if Clarise does no changes and one simply performs a translation forward and back, the resulting document will have lost a lot of information. With translations back and forth to LaTeX as well, the situation gets worse. One is effectively reduced to the intersection of the features in all the supported formats. Not much better than ASCII.

Another problem with this approach is that one needs translators for each pair of formats. For n formats we need $n(n-1)$ translators! If these are produced by different vendors, using different translation conventions, then it is even more likely that information will be lost when several are used one after another

6.3 A common format

One alternative is to look for some common format that can be used for interchange. Indeed that is one of the purposes of SGML DTDs and other standards such as ODA [9]. We then need only translators to and from each format to the common format. This format can either be a minimal format (like ASCII again!) which only supports common features, or a super, all-embracing format that has every feature that is in any of the others.

$$\begin{tikzcd}
 & \text{super format} \arrow[dl, shift left] \arrow[d, shift left] \arrow[dr, shift left] & \\
\text{RTF} \arrow[ur, shift left] \arrow[dr, shift left] & \text{SGML} \arrow[u, shift left] \arrow[d, shift left] & \text{LaTeX} \arrow[ul, shift left] \arrow[dl, shift left] \\
 & \text{ASCII} \arrow[ul, shift left] \arrow[u, shift left] \arrow[ur, shift left] &
\end{tikzcd}$$

Fig. 12. A common format

Using a super format means that only $2n$ translators are needed, and, so long as the two translators to and from each format are written properly (i.e., inverses), they are more likely to be more consistent. Also, when you translate to the super-format you no longer lose any information – great. Unfortunately, when you take documents back into a representation that can be used, oops, it's all lost again. Imagine you perform the translation:

$$\text{RTF} \longrightarrow \text{super format} \longrightarrow \text{LaTeX}$$

There is no reason to expect the result to be better than a direct RTF to LaTeX conversion. By the time you translate back and forth a few times you are back to the minimal information again.

6.4 Maintaining an invariant

What is required is a way of working such that information which has been added in one format, but which cannot be translated is recovered when the reverse translation is performed. That is, one effectively keeps a model of the document in each of the formats. When a document is changed in one format, then only the part that changes is updated in the other formats. Similar issues have been heavily studied in the context of continuously maintained mappings between the internal state of an interactive system and its display [6]. That is a functional relationship, but the principal is similar. At any time we expect that all the copies of the document in the different formats satisfy an invariant. This can most easily be expressed in terms of a common format, for example, that the ASCII version of all of them is the same.

Then, when one is updated, one looks for an equivalent update to each of the others which maintains the invariant (see figure 13).

$$\begin{tikzcd}
\text{RTF} & & \text{SGML} & & \text{LaTeX} \\
r \arrow[d, "o"'] \arrow[r, phantom, "\approx"] & & s \arrow[d, "o_s"'] \arrow[r, phantom, "\approx"] & & t \arrow[d, "o_t"'] \\
r' \arrow[r, phantom, "\approx"] & & s' \arrow[r, phantom, "\approx"] & & t'
\end{tikzcd}$$

Fig. 13. Parallel translations

Formally we can phrase this as follows. Given documents r, s and t such that $inv(r, s, t)$ and an operation o on r find operations o_s and o_t, such that $inv(o(r), o_s(s), o_t(t))$. The generation of the equivalent updates o_s and o_t will in general not be a simple function of o, but also include information stored about the documents and their relationship.

6.5 Using dynamic pointers

This is just the sort of thing dynamic pointer projections were designed for! We'll concentrate on just two of the documents r and s. We'll assume we have a translator between the document formats, which can work on fragments (with known context) – that is an incremental translator. This translator can be developed in an ad hoc fashion or be based on translation to and from a common format. In addition, we demand that these translators are projections, they supply us with a mapping between the pointers in each document format. For example, the RTF to SGML converter would have signature:

$trans \quad : Params \rightarrow (RTF \rightarrow SGML)$
$fwd_{trans}: Pt_{RTF} \rightarrow Pt_{SGML}$
$back_{trans}: Pt_{SGML} \rightarrow Pt_{RTF}$

We will use these (in a manner described below) to maintain continuously a projection between r and s. This projection is essentially the invariant (saying they have the same content) together with a pointer mapping saying which parts of the documents are equivalent.

$inv \quad : Params \rightarrow (RTF \leftrightarrow SGML)$
$fwd_{inv}: Pt_{RTF} \rightarrow Pt_{SGML}$
$back_{inv}: Pt_{SGML} \rightarrow Pt_{RTF}$

Now, given any operation o on r we can obtain its locality information loc_o. First of all, we know that any part of s which does not intersect $fwd_{inv}(loc_o)$ is unchanged by the operation. To build the rest of s', we simply use the translator on the portion of r within

loc_o. The translated bit is 'glued' into place and the new SGML document is complete. Any mark-up on the unaltered bits is preserved. The changed bits may need some additional repair by the user as they will simply have the default mark-up generated by the translator, but the effort is minimal compared to the other alternatives.

One final thread to tie up is the pointer part of the new invariant, fwd'_{inv} and $back'_{inv}$. This is repaired as follows. It is easy to build a pull function for o_s (outside of $fwd_{inv}(loc_o)$) based on the difference in and location of the updated part. The pointers outside loc_o are then related by:

$$fwd'_{inv} = pull_{o_s} \circ fwd_{inv} \circ pull_o^{-1}$$
$$back'_{inv} = pull_o \circ back_{inv} \circ pull_{o_s}^{-1}$$

This is basically shifting around the existing pointer mapping. Finally, the fwd and $back$ maps for loc_o itself are repaired by 'gluing' in the pointer map generated by fwd_{trans} and $back_{trans}$ restricted to loc_o.

6.6 Generating pull functions

The above method relies on having a pull function for update operations and 'fwd' and 'back' mappings for translations. The latter requires some rewriting of the translators, but is not too difficult. The former is more of a problem as this information would ideally be maintained by the relevant editors. However, even if Microsoft do not bring out a dynamic pointer compatible version of Word in the near future, all is not lost! The pull function can be generated by a modified file difference utility. Obviously this would need to be tuned for any particular file format, but would be useful for other purposes (such as version control) anyway.

7 Conclusions

We have seen that three different problems can all be seen as manifestations of a single phenomenon, translating operations between contexts. In each case we have needed operations which were originally formulated in one context to be used in another. The first two situations, undo and merge, were particularly similar. However, we found that there were subtle but important differences between all the translation operations. We also found that it is only necessary to define atomic translations, that is translations of single operations. Translations for sequences of operations can be generated from these atomic translations by chasing commuting diagrams.

Dynamic pointers can also be regarded as a form of translation between contexts, in this case translating locations rather than operations. However, the part of an operation which requires changing between contexts is often locational and hence dynamic pointers can be used to describe or even implement translation policies. This further eases the problem of defining suitable translations.

An important new application of dynamic pointer techniques has been described, where different representations of a document can be maintained in parallel. This offers a better hope for inter-application compatibility than the definition of common formats as it allows each to develop their own strengths, rather than being ossified in a fixed standard.

References

1. Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.
2. A. J. Dix. Dynamic pointers and threads. *Collaborative Computing (accepted for publication)*, 1994.
3. A.J. Dix. *Formal Methods for Interactive Systems*. Academic Press, 1991.
4. Alan J. Dix and Victoria C. Miles. Version control for asynchronous group work. Technical Report YCS 181, Computer Science Dept., University of York, U.K., 1992. (Poster presentation HCI'92: People and Computers VII).
5. C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. *SIGMOD Record*, 18(2):399–407, June 1989. 1989 ACM SIGMOD International Conference on Management of Data.
6. M. D. Harrison and A. J. Dix. Modelling the relationship between state and display in interactive systems. In P.Gornay and M.J.Tauber, editors, *Visualisation in Human–Computer Interaction*, volume LNCS 439, pages 241–249. Springer-Verlag, 1990.
7. J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.
8. Atul Prakash and Michael J. Knister. Undoing actions in collaborative work. In *CSCWU92 – Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 273–280. ACM Press, 1992.
9. A. Schill. *Cooperative Office Systems*. Prentice Hall, 1995.
10. P. Wright, A. Monk, and M. Harrison. State, display an undo: a study of consistency in display base interaction. Technical report, University of York, 1992.

Appendix 1 – Differences between transations

In section 3.3, we discussed several putative identites which look as though they might hold between undo and merge translations. We now give the counter examples showing that the translations cannot easily be obtained from one another by use of simple identities. The identities we proposed based on comparing the two commuting diagrams were were:

(i) $\overleftarrow{(\underrightarrow{\mathrm{b}}_{\mathrm{a}})}^{\mathrm{a}} = b$

(ii) $\underrightarrow{(\overleftarrow{\mathrm{b}}^{\mathrm{a}})}_{\mathrm{a}} = b$

(iii) $\overrightarrow{\mathrm{a}}^{(\underrightarrow{\mathrm{b}}_{\mathrm{a}})} = \underleftarrow{\mathrm{a}}_{\mathrm{b}}$

(iv) $\underleftarrow{\mathrm{a}}_{(\overleftarrow{\mathrm{b}}^{\mathrm{a}})} = \overrightarrow{\mathrm{a}}^{\mathrm{b}}$

Figures 6 and 7, showed the problematic insert–insert and delete–insert diagrams. We use these to build the counter-examples. As sort hand, we will write 'dn' for $delete(n)$ and 'dn+1' for $delete(n + 1)$. Similarly, we will use 'in', 'in-1' an 'in+1' for insertions. The character to insert is obvious from context and is irrelevant in the examples.

Insert–insert cases

We need to look at two instances of undo translations (note that these are laid out to resemble the diagram order):

(u1) a = in $\qquad$ b = in

$\overleftarrow{\text{b}}^{\text{a}}$ = in $\qquad$ $\overrightarrow{\text{a}}^{\text{b}}$ = in+1

(u2) a = in+1 $\qquad$ b = in

$\overleftarrow{\text{b}}^{\text{a}}$ = in $\qquad$ $\overrightarrow{\text{a}}^{\text{b}}$ = in

There is only one merge rule (which is precisely the problem!). Recall that there were two possible rules for merging two inserts at the same location. The alternative rule is shown in brackets. We will track it also as we follow through the examples, an we will see that whichever merge rule is adopted counter-examples can be found.

(m1) a = in $\qquad$ $\underrightarrow{\text{b}}_{\text{a}}$ = in [in+1]

b = in $\qquad$ $\underleftarrow{\text{a}}_{\text{b}}$ = in+1 [in]

In section 3.3, we said that (i) and (iii) are true for the insert–insert scenario. This can easily be verified. So, we will look at (ii) and (iv). First for the case with a = 'in' and b = 'in+1':

(ii) $\underrightarrow{(\overleftarrow{\text{in+1}}^{\text{in}})}_{\text{in}} = \underrightarrow{\text{in}}_{\text{in}}$

= in – NO [in+1 – OK]

(iv) RHS: $\overrightarrow{\text{in}}^{\text{in+1}}$ = in

LHS: $\underleftarrow{\text{in}}_{(\overleftarrow{\text{in+1}}^{\text{in}})} = \underleftarrow{\text{in}}_{\text{in}}$

= in+1 – NO [in – OK]

These counter-examples seem to favour the alternative merge rules, but if we look at the case with both a and b equal to 'in' we see that this rule doesn't work either.

(ii) $\underrightarrow{(\overleftarrow{\text{in}}^{\text{in}})}_{\text{in}} = \underrightarrow{\text{in}}_{\text{in}}$

= in – OK [in+1 – NO]

(iv) RHS: $\overrightarrow{\text{in}}^{\text{in}}$ = in+1

LHS: $\underleftarrow{\text{in}}_{(\overleftarrow{\text{in}}^{\text{in}})} = \underleftarrow{\text{in}}_{\text{in}}$

= in+1 – OK [in – NO]

So, whichever merge rule is chosen, neither identity holds in general.

Delete–insert cases

This time we need consider only one instance of the undo rule. However, like the case of the merge rule above, it is not clear which of two options should be chosen. This is because it is unclear whether an insertion following a delete belongs just before or after the deleted characters. We will again trace through both alternatives.

(u3) a = dn $\qquad$ b = in-1

$\overleftarrow{\text{b}}^{\text{a}}$ = in [in+1] $\qquad$ $\overrightarrow{\text{a}}^{\text{b}}$ = dn [dn+1]

In counter-symmetry to the insert–insert scenario, we now hae two merge cases. Happily, there are no different alternatives for these!

(m2) a = dn $\qquad$ $\underrightarrow{\text{b}}_{\text{a}}$ = in-1

b = in $\qquad$ $\underleftarrow{\text{a}}_{\text{b}}$ = dn

(m3) a = dn $\qquad$ $\underrightarrow{\text{b}}_{\text{a}}$ = in-1

b = in-1 $\qquad$ $\underleftarrow{\text{a}}_{\text{b}}$ = dn+1

This time identities (ii) and (iv) are OK, but (i) and (iii) will fail. Again, we will consider two cases, first a = ‘dn’ and b = ‘in-1’:

(i) $\overleftarrow{(\text{in-1}_{\overrightarrow{\text{dn}}})}^{\text{dn}} = \overleftarrow{\text{in-1}}^{\text{dn}}$

= in – NO [in-1 – OK]

(iii) RHS $\text{dn}_{\overleftarrow{\text{in-1}}}$ = dn+1

LHS $\overrightarrow{\text{dn}}^{(\text{in-1}_{\overrightarrow{\text{dn}}})} = \overrightarrow{\text{dn}}^{\text{in-1}}$

= dn – NO [dn+1 – OK]

Again, this case seems to favour the alternative for undo. But, now consider the case where a = ‘dn’ and b = ‘in’:

(i) $\overleftarrow{(\text{in}_{\overrightarrow{\text{dn}}})}^{\text{dn}} = \overleftarrow{\text{in-1}}^{\text{dn}}$

= in – OK [in-1 – NO]

(iii) RHS $\text{dn}_{\overleftarrow{\text{in}}}$ = dn

LHS $\overrightarrow{\text{dn}}^{(\text{in}_{\overrightarrow{\text{dn}}})} = \overrightarrow{\text{dn}}^{\text{in-1}}$

= dn – OK [dn+1 – NO]

So, whichever alternative we choose for the undo rule, both (i) an (iii) are invalid.

Appendix 2 – Soundness of compositon laws

In section 4, we looked at sequence definitions of the various forward and backward translation operators. These allowed us to extend the atomic translation definitions to work on sequences of operations. These definitions are intended to work for any atomic translations satisfying the basic translation laws:

$ab = \overleftarrow{b}^{a}\overrightarrow{a}^{b}$ $\qquad a\, b_{\overrightarrow{a}} = b\, a_{\overleftarrow{b}}$

We noted that it is important that the composition rules are sound, in the sense that all rewrites of an expression using the rules get to the same result, a confluence property. This will ensure that the sequence rules are conservative, that is they do not give rise to additional equations for the atomic translations. We will show that this is true for the undo laws. The proof for merge is similar.

Recall that the laws for undo are:

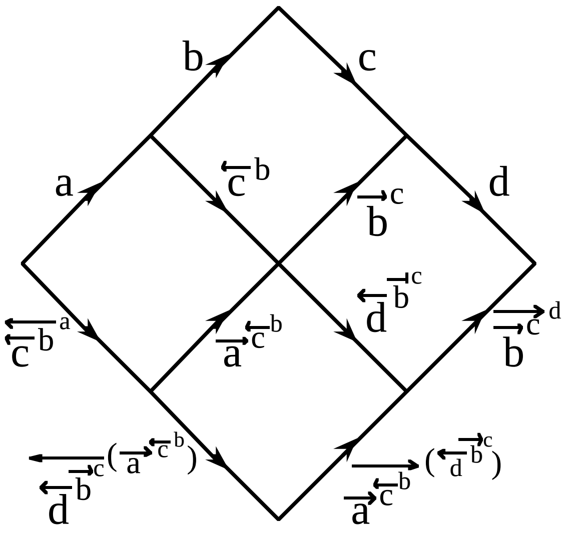
(i) $\overrightarrow{a}^{(bc)} = \overrightarrow{(\overrightarrow{a}^{b})}^{c}$ $\qquad$ (iii) $\overrightarrow{ab}^{c} = \overrightarrow{a}^{(\overleftarrow{c}^{b})}\overrightarrow{b}^{c}$

(ii) $\overleftarrow{bc}^{a} = \overleftarrow{b}^{a}\overleftarrow{c}^{(\overrightarrow{a}^{b})}$ $\qquad$ (iv) $\overleftarrow{c}^{(ab)} = \overleftarrow{(\overleftarrow{c}^{b})}^{a}$

The only way rewrites can fail to be confluent is when there are two ways of reducing the outermost operator. In the case of the above laws, this can only happen in the cases:

$\overrightarrow{ab}^{(cd)}$ and $\overleftarrow{cd}^{ab}$

Concentrating on the first, this can be reduced in two ways. Either by (i) or by (iii). We show that both lead to the same cannonical form.



a
b
c
d
$\overleftarrow{c}^{b}$
$\overrightarrow{b}^{c}$
$\overleftarrow{\overleftarrow{c}^{b}}^{a}$
$\overrightarrow{a}^{\overleftarrow{c}^{b}}$
$\overleftarrow{d}^{\overrightarrow{b}^{c}}$
$\overrightarrow{\overrightarrow{b}^{c}}^{d}$
$\overleftarrow{\overleftarrow{d}^{\overrightarrow{b}^{c}}}^{(\overrightarrow{a}^{\overleftarrow{c}^{b}})}$
$\overrightarrow{\overrightarrow{a}^{\overleftarrow{c}^{b}}}^{(\overleftarrow{d}^{\overrightarrow{b}^{c}})}$

Fig. 14. Checking correctness of compositional definitions

$$\overrightarrow{ab}^{(cd)} = \text{(i)}\ \overrightarrow{\overrightarrow{ab}^{c}}^{d} = \text{(iii)}\ \overrightarrow{\left(\overrightarrow{a}^{(\overleftarrow{c}^{b})}\ \overrightarrow{b}^{c}\right)}^{d}$$

$$= \text{(iii)}\ \overrightarrow{\overrightarrow{a}^{(\overleftarrow{c}^{b})}}^{\overleftarrow{d}^{(\overrightarrow{b}^{c})}}\ \overrightarrow{\overrightarrow{b}^{c}}^{d}$$

$$= \text{(iii)}\ \overrightarrow{a}^{(\overleftarrow{cd}^{b})}\ \overrightarrow{b}^{cd} = \text{(ii)}\ \overrightarrow{a}^{(\overleftarrow{c}^{b}\,\overleftarrow{d}^{(\overrightarrow{b}^{c})})}\ \overrightarrow{b}^{cd}$$

$$= \text{(iii)}\ \overrightarrow{\overrightarrow{a}^{(\overleftarrow{c}^{b})}}^{\overleftarrow{d}^{\overrightarrow{b}^{c}}}\ \overrightarrow{\overrightarrow{b}^{c}}^{d}$$

The second case is similar. It can be reduced by (ii) or (iv), but again both reduce to the same cannonical form.

$$\begin{aligned}
\overleftarrow{cd}^{(ab)} &= \text{(ii)}\ \overleftarrow{c}^{ab}\,\overleftarrow{d}^{(\overrightarrow{ab}^{c})} &&= \text{(iii)}\ \overleftarrow{c}^{ab}\,\overleftarrow{d}^{(\overrightarrow{a}^{(\overleftarrow{c}^{b})}\,\overrightarrow{b}^{c})} \\
& &&= \text{(iv)}\ \overleftarrow{\overleftarrow{c}^{b}}^{a}\,\overleftarrow{\overleftarrow{d}^{(\overrightarrow{b}^{c})}}^{\overrightarrow{a}^{(\overleftarrow{c}^{b})}} \\
&= \text{(iv)}\ \overleftarrow{\overleftarrow{cd}^{b}}^{a} &&= \text{(ii)}\ \overleftarrow{\left(\overleftarrow{c}^{b}\,\overleftarrow{d}^{(\overrightarrow{b}^{c})}\right)}^{a} \\
& &&= \text{(ii)}\ \overleftarrow{\overleftarrow{c}^{b}}^{a}\,\overleftarrow{\overleftarrow{d}^{(\overrightarrow{b}^{c})}}^{\overrightarrow{a}^{(\overleftarrow{c}^{b})}}
\end{aligned}$$

The intermediate results of these two proofs are shown on the commuting diagram in figure 14. The proof for the merge case is similar, but slightly simpler as the two merge translations are duals of one another and hence only one case need be considered.