Operational transform
Merging operations

Sep 2005
David Barrett-Lennard

Abstract
This paper investigates merging of operations in such a way that inclusion transformation and exclusion transform are still supported, making it useful for overcoming the inherent quadratic complexity of inclusion transforming two lists of operations.

Introduction
In order for operational transform to be successfully applied to configuration management, it is necessary to efficiently merge branches containing hundreds of check-ins. Unfortunately, inclusion transform of two branches containing n_1 and n_2 operations involves $n_1 n_2$ transformations which is prohibitive.

This paper assumes that each branch can be compressed into a single, concise state difference, and the brute force IT of the two branches is equivalent to IT of the two state differences.

For a text document, a state difference is represented as a set of extractions followed by a set of insertions. Merging, IT and ET are all implemented in linear time using a left to right scan of the intervals (see [1] and [2]). However the merging algorithm in [1] is incompatible with subsequent inclusion transform in [2] because identity of insertions and deletions is lost. Furthermore [1] doesn't account for the q-position.

IT/ET has been solved for assignment operations [3]. This paper investigates merging of assignment operations.

Notational conventions
For state S and operation O, S' = S+O denotes the state obtained after executing O on state S. Operation O may only be executed on state S. Therefore we define $state_{in}(O) = S$.

Let id(O) denote the site identifier of the site on which O was originally generated. It is assumed there is a total ordering on site identifiers.

Two operations O_1, O_2 are *equivalent* (written $O_1 \sim O_2$) if $state_{in}(O_1) = state_{in}(O_2) = S$ and $S+O_1 = S+O_2$. Note that this doesn't imply that the operations are equal. For example, it is possible that $id(O_1) \neq id(O_2)$.

$[O_1 \ldots O_n]$ denotes the list of operations $O_1,\ldots,O_n$ assumed to be contextually serialised - ie intended to be performed in the given order on some initial state S. $S + [O_1 \ldots O_n]$ denotes the state $(((S+O_1)+O_2)+\ldots+O_n)$. [] denotes an empty list.

It is convenient to let $[X_1, \ldots, X_n]$ represent the list which concatenates the X_i which can be any mixture of lists or individual operations. For example $[L_1, L_2, O_3]$ concatenates list L_1, list L_2 and operation O_3 into a single linear list of operations.

Inclusion transform
Let $O_1 \parallel O_2$ and $state_{in}(O_1) = state_{in}(O_2) = S$. Then we define $O_1' = IT(O_1,O_2)$ as being a transformed version of O_1 that maintains the original intention of O_1, whilst being executed in the document state following execution of O_2. ie $state_{in}(O_1') = S+O_2$. Therefore $[O_2\ IT(O_1,O_2]$ is contextually serialised.

List properties of IT

Let L_1, L_2, L_3 be lists of operations. We require the following three properties

LIT1 $IT([], []) = []$

LIT2 $IT(L_1, [L_2 , L_3]) = IT(IT(L_1, L_2), L_3)$

LIT3 $IT([L_1, L_2], L_3) = [\ IT(L_1,L_3)\ ,\ IT(L_2, IT(L_3, L_1))\ ]$

It can be shown that $IT([], L) = []$, and $IT(L, []) = L$.

These properties can be regarded as a definition of $IT(L_1, L_2)$. Note that this definition appears to be ambiguous in the way in which a list is decomposed into smaller sub-lists. However, it turns out that the final result is independent of the order in which the lists are decomposed.

Transformation properties of IT

TP1 $S + [O_1\ IT(O_2, O_1)] = S + [\ O_2\ IT(O_1, O_2)\ ]$

TP2 $IT(\ IT(\ O_3, O_1),\ IT(O_2, O_1)) = IT(\ IT(\ O_3, O_2),\ IT(O_1, O_2))$

Merging operations

Given contextually serialised operations $[O_1, O_2]$, we define $O_1 \oplus O_2$ as the merge of O_1, O_2 into a single composite operation. Therefore the properties above also apply to $O_1 \oplus O_2$ as it it represents a single operation. For example $IT([], O_1 \oplus O_2) = []$, and $IT(O_1 \oplus O_2, []) = O_1 \oplus O_2$

Note that for contextually serialised operations $[O_1, O_2]$, $O_2 \oplus O_1$ is undefined.

Lists of operations can't be directly merged. ie $L_1 \oplus L_2$, $L \oplus O$ and $O \oplus L$ are undefined.

We require the following properties.

MP1 $(O_1 \oplus O_2) \oplus O_3 = O_1 \oplus (O_2 \oplus O_3)$ (associativity)

MP2 $S + [O_1\ O_2] = S + (O_1 \oplus O_2)$

MP3 $IT(O_1, O_2 \oplus O_3) = IT(O_1, [O_2, O_3])$

MP4 $IT(O_1 \oplus O_2, O_3) = IT(O_1,O_3) \oplus IT(O_2, IT(O_3, O_1))$

MP1 ensures we can merge operations in any order

MP2 ensures that merging operations doesn't affect the final document state.

MP3 ensures that merging doesn't change the effect on an operation as it IT's past the merged operations.

MP4 ensures that when a list of operations is IT'd past an operation, merging can be performed before or after the IT.

Given list L, let $\sum L$ represent the result of merging all operations in L into a single composite operation. The order in which adjacent operations are merged doesn't matter by property MP1 (associativity of $\oplus$). We define $\sum[] = []$ and $\sum[O] = O$.

Lemma 1: $IT(L, O_1 \oplus O_2) = IT(L, [O_1, O_2])$
Proof: (by induction on n = |L|)
n=0: $IT([], O_1 \oplus O_2) = [] = IT([], [O_1, O_2])$
n=1: Follows directly from MP3
n→n+1: Let L' = [O,L]. Then $IT(L', O_1 \oplus O_2)$

$= IT([O,L], O_1 \oplus O_2)$
$= [IT(O, O_1 \oplus O_2) , IT(L, IT(O_1 \oplus O_2, O))]$ (LIT3)
$= [IT(O, [O_1,O_2]) , IT(L, IT(O_1 \oplus O_2, O))]$ (MP3)
$= [IT(O, [O_1,O_2]) , IT(L, IT(O_1,O) \oplus IT(O_2, IT(O, O_1)))]$ (MP4)
$= [IT(O, [O_1,O_2]) , IT(L, [IT(O_1,O) , IT(O_2, IT(O, O_1))])]$ (inductive step)
$= [IT(O, [O_1,O_2]) , IT(L, IT([O_1,O_2], O))]$ (LIT3)
$= IT([O,L], [O_1,O_2])$ (LIT3)
$= IT(L', [O_1,O_2])$

Lemma 2: $IT(L, [L_1,O_2,O_3,L_4]) = IT(L, [L_1, O_2 \oplus O_3, L_4])$
ie When IT'ing past a list, we can merge any adjacent operations in the list without affecting the result.
Proof: $IT(L, [L_1,O_2,O_3,L_4])$

$= IT(IT(IT(L, L_1), [O_2,O_3]), L_4])$ (LIT2)
$= IT(IT(IT(L, L_1), O_2 \oplus O_3), L_4])$ (Lemma1)
$= IT(L, [L_1, O_2 \oplus O_3, L_4])$ (LIT2)

By repeated application of this lemma we deduce that $\forall L_1, L_2,\ IT(L_1, \sum L_2) = IT(L_1, L_2)$

Lemma 3: $IT(O_1 \oplus O_2, L) = IT(O_1,L) \oplus IT(O_2, IT(L, O_1))$
This is an extension to MP4.
Proof : (by induction on n = |L|)
n=0: $IT(O_1 \oplus O_2, []) = O_1 \oplus O2 = IT(O_1,[]) \oplus IT(O_2, IT([], O_1))$
n=1: Follows directly from MP4
n→n+1: Let L' = [O,L]. Then $IT(O_1 \oplus O_2, L')$

$= IT(O_1 \oplus O_2, [O,L])$
$= IT(IT(O_1 \oplus O_2, O) , L)$ (LIT2)
$= IT(IT(O_1,O) \oplus IT(O_2, IT(O, O_1)) , L)$ (MP4)
$= IT(IT(O_1,O),L) \oplus IT(IT(O_2, IT(O, O_1)) , IT(L, IT(O_1,O)))$ (inductive step)
$= IT(O_1, [O,L]) \oplus IT(O_2, [IT(O, O_1) , IT(L, IT(O_1, O))])$ (LIT2)
$= IT(O_1, [O,L]) \oplus IT(O_2, IT([O,L], O_1))$ (LIT3)
$= IT(O_1,L') \oplus IT(O_2, IT(L', O_1))$

Lemma 4: $IT(\sum L_1, L_2) = \sum IT(L_1, L_2)$
Proof : (by induction on n = $|L_1|$)
n=0: $IT(\sum L_1, L_2) = IT(\sum[], L_2) = IT([], L_2) = [] = \sum[] = \sum IT([], L_2) = \sum IT(L_1, L_2)$
n=1: $IT(\sum L_1, L_2) = IT(\sum[O], L_2) = IT(O, L_2) = \sum IT([O], L_2) = \sum IT(L_1, L_2)$
n→n+1: Let $L_1' = [O,L_1]$. Then $IT(\sum L_1', L_2)$

$= IT(\sum [O,L_1] , L_2)$
$= IT(O \oplus \sum L_1 , L_2)$
$= IT(O, L_2) \oplus IT(\sum L_1, IT(L_2, O))$ (Lemma 3)
$= IT(O,L_2) \oplus \sum IT(L_1, IT(L_2,O))$ (inductive step)

$= \sum$ [$IT(O,L_2)$, $IT(L_1, IT(L_2,O))$]
$= \sum IT([O,L_1] , L_2)$
$= \sum IT(L_1', L_2)$

TODO: Need to show that merging is compatible with achieving convergence at all sites.

Merging in practice
In [2] and [3], a comparison of site identifiers (which are assumed to be in a total order) is required to break symmetry in certain cases of operational transform, to allow all sites to converge at quiescence. This shows that site identifiers will need to be stored when merging operations.

In [1], a multi-char insert/delete operation stores a set of extraction intervals and insertion intervals. Each interval stores a string, p-position and q-position. Consider that it also stores a site identifier. In practice a site identifier needs to be fairly large in order to support global uniqueness. For example, a 128 bit GUID is suitable for this purpose. For space efficiency a site could use a local map from a 32 bit identifier to the 128 bit GUID. Therefore the overhead per interval is insignificant.

Note that the generating sequence number of the operation is not needed for operational transform, so this value doesn't need to be stored in the operation.

The result is that merging is able to compress edits by a given user, but not between users. This will still allow good compression because users tend to localise their work. A useful side effect is that any state difference returned by a repository will have all the information about what user did what.

Assignment operations
Assignment operations make use of a q-position - because an assignment is regarded as an insertion in the effects document. So under merging it is necessary to keep track of what was inserted where in the effects document. Therefore to support merging an assignment operation needs to store a set of insertion intervals, located by their q-position. It doesn't seem necessary to store all the values to be inserted in the effects document, apart from left most insertion by the operation, because we are never actually interested in the effects document apart from the value at position 0, and of course correctly transforming q-positions.

Note that we expect ET to work correctly with merged operations, making it clear that q-positions of assignments must be transformed as though no merging had taken place.

Problem
Consider that we compress a list of string-wise delete/insert operations. We must support compression of causally dependent operations. Therefore it is possible that an operation O_1 inserts characters that are subsequently deleted by an operation O_2.

We have assumed that after merging all operations we end up with a single set of deletion intervals, following by a single set of insertion intervals. This representation makes it *impossible* to represent the characters that were inserted by O_1 then deleted by O_2, because the characters aren't available to be deleted in the initial document state This suggests that merging can't satisfy properties MP3, MP4.

One possibility could be to represent operations as a set of insertions following by a set of deletions! Now there is no difficulty representing characters that are inserted then subsequently deleted. A downside is that this doesn't relate easily to a move operation - because it is strange to insert before extracting. Nevertheless, this approach is promising enough to investigate more fully.

Merging is achieved as follows : $[I_1\ X_1\ I_2\ X_2] \rightarrow [I_1\ I_2'\ X_1'\ X_2] \rightarrow [I\ X]$. The initial transpose is different than the current approach.

It should still be possible to transpose operations, and dual IT operations. The order in which the steps are performed is different. However, the existing functions should still work - it is simply a matter of calling them in a different order.

References

[1]	David Barrett-Lennard. Log Compression Algorithm. July 2005.
[2]	David Barrett-Lennard. Operational transform - Multi-char insert and delete operations. Aug 2005
[3]	David Barrett-Lennard. Operational transform - Assignment operations. Aug 2005