Single character move, insert and delete operations

David Barrett-Lennard
20 April 2007

Introduction

In this paper we provide an algorithm for operational transform on single character move, insert and delete operations on text documents.

Holding areas

A possible approach is to only use move operations and use explicit documents with identity called *holding areas* to emulate pure insertion or pure deletion. For example a move from a holding area into a document simulates a pure insertion. As far as the documents are concerned matter has been created from nowhere even though the underlying move operations always conserve matter.

This would appear to simplify the problem. However there are two issues

1. The overhead of lots of tiny, one character holding areas generated while editing text could be significant.
2. It is far from clear how log compression can be achieved.

Instead we assume operations must directly represent pure deletes and inserts (in addition to moves).

Effects document

An *effects document* is a hypothetical document in which characters are inserted but never removed. As a result move operations behave identically to insert operations and delete operations have no effect.

It is assumed there are a set of effects documents, each uniquely identified by some document id.

For the purposes of this exposition we assume the following simple implementation (given in C++) of a set of effects documents:-

```
struct Entry
{
    char value;
    bool present;
    bool deleted;
};
typedef vector<Entry> EffectsDoc;
typedef vector<EffectsDoc> EffectsDocSet;
typedef int DocId;
```

The meaning and purpose of the present and deleted flags is described in the following sections.

Working document

The working document corresponds to the document state as understood by an end user, and can be derived from the corresponding effects document by simply filtering out the characters that don't *exist*, according to the following definition

```
exists(c) =  c.present && !c.deleted
```

Insert operation

An insert operation is assumed to insert a single character into an effects document.

```
struct InsertOp
{
    SiteId id;  // Identifies site that generated the operation
    DocId di;   // Destination document
```

```
    int dq;     // Destination position
    char c;     // Character being inserted
};
```

This inserts character c at position dq in document di.

More specifically it inserts a character entry into the effects document di at position dq with

```
    value = c;
    present = true;
    deleted = false;
```

Insert operations can never be disabled under IT. There is no tracking rule. The insertion q-position must be shifted as required but will never track into a different document.

Move operation

A move operation contains the following state

```
struct MoveOp
{
    SiteId id;  // Identifies site that generated the operation
    int e;      // Enable status
    DocId si;   // Source document
    int sq;     // Source position
    DocId di;   // Destination document
    int dq;     // Destination position
    char c;     // Character being moved
};
```

It is assumed that the source position is in post insertion coordinates. This is significant if the destination position is to the left of the source position.

Each character location in an effects document carries a flag to mark whether the character is currently *present* at that location. When a character is moved (perhaps to a different effects document), a character of the same value is inserted at the destination position of the move operation where it is marked as present and the original character location (ie the source of the move) is marked as no longer present. Over time a character may be moved many times, causing it to appear in many different locations across the set of all effects documents. However at any given time, there will be precisely one location of the character where it is marked as present. Note in particular that as far as the present flag is concerned a character can never be removed from the set of effects documents once it has been inserted. Furthermore it is always present at one well defined location. So we see that move operations conserve matter – in the sense of the set of characters that are marked as present.

Note that once the present flag has been cleared on a given character location, the flag can never be set again. It is understood that a move operation will always insert a new character into an effects document – even to emulate the effect of moving a character back to a previous location in the working document.

Under IT there is a requirement that the source position of a move operation always track the one and only current location of the character being moved, irrespective of whether the move operation is enabled or not.

The enable status ‘e’ is used to pick a unique winner amongst all competing move operations that have tried to move the same character to different locations. As discussed in earlier work, e represents an insertion position in a hypothetical effects document, and the unique winner is defined to be the one and only move operation that is inserting at position e = 0.

Delete operation

```
struct DeleteOp
{
    SiteId id;  // Identifies site that generated the operation
    DocId si;   // Source document
    int sq;     // Source position
    char c;     // Character being deleted
};
```

Each character in an effects document contains a flag ‘deleted’ to indicate whether it has been (permanently) deleted. It is important to understand that this flag is distinct from the ‘present’ flag. In fact, it will commonly be the case that a character will be marked as present and yet deleted.

A delete operation never clears the present flag of a character. Instead it only sets the deleted status. As a result delete operations don’t have any impact on move or insert operations under IT.

Similarly to a move operation, the source position of a delete operation is required to track the one and only current location of the character that is being deleted (ie where the current location is defined by the present status).

There is no concept of an undelete operation. Once a character has been deleted there is no way to bring it back. Note as well that deletion only involves setting the delete status and that is idempotent. Therefore a character can be deleted any number of times without affecting the fact that it has been deleted.

The deleted status that is flagged in the effects document must only be set at the location of the character where it is currently present. Note therefore that when a move operation is applied it may be necessary for the deleted status to be moved together with the present status.

Operation

For the convenience of this exposition, we assume that a single data structure is used for all three types of operation as follows

```
enum Type
{
    MOVE,
    INSERT,
    DELETE
};

struct Operation
{
    SiteId id;  // Identifies site that generated the operation
    Type type;  // Type of operation
    int e;      // Enable status
    DocId si;   // Source document
    int sq;     // Source position
    DocId di;   // Destination document
    int dq;     // Destination position
    char c;     // Character being moved/inserted/deleted
};
```

Applying an operation

The following C++ implementation shows the required changes to a given EffectsDocSet as a result of applying an operation.

```
void ApplyOperation(const Operation& o, EffectsDocSet& ds)
{
```

```
    if (o.type == INSERT)
    {
        ds[o.di].insert(o.dq);
        ds[o.di][o.dq].value = o.c;
        ds[o.di][o.dq].present = true;
        ds[o.di][o.dq].deleted = false;
    }
    else if (o.type == DELETE)
    {
        ds[o.si][o.sq].deleted = true;
    }
    else
    {
        // Move operation
        ds[o.di].insert(o.dq);
        ds[o.di][o.dq].value = o.c;
        ds[o.di][o.dq].present = (o.e == 0);
        ds[o.di][o.dq].deleted = false;
        if (o.e == 0 && ds[o.si][o.sq].deleted)
        {
            ds[o.si][o.sq].deleted = false;
            ds[o.di][o.dq].deleted = true;
        }
    }
}
```

IT and ET

The following are the implementation of DualIT and Transpose

```
// O1, O2 are concurrent context equivalent operations
// IT O1 past O2 and O2 past O1
void DualIT(Operation& O1, Operation& O2)
{
    if (O1.type != DELETE && O2.type != DELETE && O1.di == O2.di)
    {
        if (O2.dq < O1.dq || O2.dq == O1.dq && O2.id < O1.id) ++O1.dq; else ++O2.dq;
    }
    if (O1.type != DELETE && O2.type != INSERT && O1.di == O2.si && O1.dq <= O2.sq) ++O2.sq;
    if (O2.type != DELETE && O1.type != INSERT && O2.di == O1.si && O2.dq <= O1.sq) ++O1.sq;
    if (O1.type != INSERT && O2.si != INSERT && O1.si == O2.si && O1.sq == O2.sq)
    {
        if (O1.type == MOVE && O1.e == 0) { O2.si = O1.di; O2.sq = O1.dq; }
        if (O2.type == MOVE && O2.e == 0) { O1.si = O2.di; O1.sq = O2.dq; }
        if (O1.type == MOVE && O2.type == MOVE)
        {
            if (O2.e < O1.e || O2.e == O1.e && O2.id < O1.id) ++O1.e; else ++O2.e;
        }
    }
}

// O1,O2 are concurrent contextually serialised operations [O1 O2]
// Transpose their contextual order to [O2 O1]
void Transpose(Operation& O1, Operation& O2)
{
    int prev2dq = O2.dq;
    if (O2.type != DELETE && O1.type != INSERT && O2.di == O1.si && O2.dq <= O1.sq) ++O1.sq;
    if (O1.type != DELETE && O2.type != DELETE && O1.di == O2.di)
        { if (O1.dq < O2.dq) --O2.dq; else ++O1.dq; }
    if (O1.type != INSERT && O2.type != INSERT && O1.si == O2.si && O1.sq == O2.sq ||
        O1.type != DELETE && O2.type != INSERT && O1.di == O2.si && O1.dq == O2.sq)
    {
        if (O1.type == MOVE && O1.e == 0) { O2.si = O1.si; O2.sq = O1.sq; }
        if (O1.type == MOVE && O2.type == MOVE) { if (O1.e < O2.e) --O2.e; else ++O1.e; }
        if (O2.type == MOVE && O2.e == 0) { O1.si = O2.di; O1.sq = prev2dq; }
    }
    if (O1.type != DELETE && O2.type != INSERT && O1.di == O2.si && O1.dq < O2.sq) --O2.sq;
}
```