Repository - Assignable Fields

Feb 2008
David Barrett-Lennard

Introduction

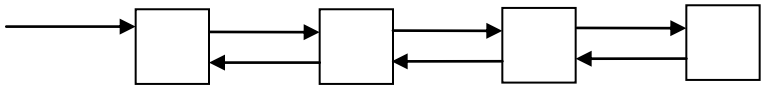
The document concerns the representation within a ceda repository of all checked in assignments to a single assignable field of a single object.

Agreeing on a unique winner

Conceptually every assignment is associated with the insertion of a single character into some hypothetical text document at a particular q-position. The purpose is only to define a unique winner amongst competing assignment operations. Basically the one and only assignment associated with the character at q = 0 is deemed to be the winner. Of course this depends on the vector time.

Representation

The ceda repository records the assignments in a double linked list as follows



Each box in this figure represents a single assignment to the field. The following information is recorded

s	the site identifier where the assignment operation was originally generated
t	the sequence number allocated to the operation on the site where it was originally generated
value	the value to be assigned

There is a total ordering on the assignments (according to the so called *effects relation*), so there is no ambiguity in the order of these assignments in the double linked list.

Let T denote the type of the field. We could store the assignments on the field in the repository as follows

```
struct ReposAssignment
{
    SiteId s;
    int t;
    T value;
};

typedef list<ReposAssignment> ReposAssignableField;
```

Representation of an operation

The following is a suitable representation for an operation that supports information preservation under merging of assignments to a given field of type T:

```
struct SingleAssignmentOp
{
    SiteId s;
    int t;
    T value;
    int q;
};

struct MultiAssignmentOp
{
    list<SingleAssignmentOp> L;
    T prevValue;
};
```

This representation allows for a composite assignment operation to record the original (s,t) on the individual assignments. It is assumed that the list is sorted by q-position, and the q-positions are in *post insertion coordinates*.

Effects relation

Definition: We write $O_1 <_e O_2$ if the assignment O_1 dominates O_2 in all document states where both O_1 and O_2 have been executed. This is a total ordering on all the assignment operations. The assignment operations recorded in this repository are uniquely ordered according to this *effects relation*.

ie $O_1 <_e O_2 \quad \Leftrightarrow \quad O_1$ appears on left of O_2 in the repository.

Calculating the state for a given vector time

For vector time v, consider that in the linked list we remove the assignments that are not in $\chi(v)$. Then we see a sequence of assignments that have q-positions 0,1,2,... in the state corresponding to the vector time.

Therefore for vector time v, the value of the field can be found by iterating through the assignments from left to right until finding the left most assignment satisfying $t < v(s)$.

Let T represent the type of the field, and we support the forwards iteration through the assignments. Then the following idea illustrates the algorithm to calculate the value of the field for a given vector time.

```
T GetFieldValue(ReposAssignableField& f, VectorTime v)
{
    for each i in f
    {
        if (i.t < v(i.s)) return i.value;
    }
    return T();
}
```

Calculating the difference between two given vector times

Let vector times v_1, v_2 be given satisfying $v_1 \leq v_2$. We require a function that represents the state difference $\chi(v_2) \setminus \chi(v_1)$ as an operation. This operation is assumed to be applied in the context of $\chi(v_1)$ to transition the state to one that corresponds to $\chi(v_2)$.

Since the execution context for this operation is $\chi(v_1)$, we see that the previous value of the field equals GetFieldValue(v_1).

```
MultiAssignmentOp GetDiff(ReposAssignableField& f, VectorTime v1, VectorTime v2)
{
    MultiAssignmentOp o;
    o.prevValue = GetFieldValue(v1);
    int q = 0;
    for each r in f
    {
        if (r.t < v2(r.s))
        {
            if (r.t >= v1(r.s))
            {
                o.L += Interval(r.s, r.t, r.value, q);
            }
            ++q;
        }
    }
    return o;
}
```

Check-In

Consider the check-in of operation O which is a `SingleAssignmentOp`, with execution context given by vector time v = ec(O). ie O is assumed to be executed in the context of executing all operations in $\chi(v)$.

For the given vector time v we can distinguish between assignments in $\chi(v)$ versus assignments that are not in $\chi(v)$. As an example, suppose we want to check-in O with O.q = 1, and the assignments in the repository ordered by the effects relation can be depicted as follows

$$[\ x_0\ \ x_1\ \ q_0\ \ x_2\ \ x_3\ \ x_4\ \ q_1\ \ q_2\ \ q_3\ \ x_5\ \ x_6\ \ q_4\ ]$$

where the q_i depict assignments in $\chi(v)$ and the x_i depict assignments not in $\chi(v)$.

Claim: for each i, $O \parallel x_i$.
Proof:

$x_i \notin \chi(v)$	(distinction between x_i and q_i)
$x_i \notin \chi(ec(O))$	(because v = ec(O))
$\neg(x_i \rightarrow O)$	(contrapositive of $x_i \rightarrow O \Rightarrow x_i \in \chi(ec(O))$ from [1])
Also $\neg(O \rightarrow x_i)$	(x_i checked in before O, so $O \notin \chi(gc(x_i))$)
So $O \parallel x_i$	

It follows therefore that for each i, $O.s \neq x_i.s$.

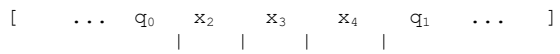
We interpret the insertion position O.q = 1 with respect to the index positions of the assignments with all the x_i removed. ie

$$[\ q_0\ \ q_1\ \ q_2\ \ q_3\ \ q_4\ ]$$

insertion with O.q = 1 must be in here

This implies that O must be inserted after q_0 but before q_1.

In the original full list of assignments, we actually have x_2,x_3,x_4 between q_0,q_1. Therefore there are four possible insertion positions:



In turns out there is a unique position where O must be inserted. Let the z_i refer to x_2,x_3,x_4 in this example.

Suppose firstly that the z_i are mutually concurrent assignments. ie for each i,j $z_i \parallel z_j$. This implies they were generated on different sites. Furthermore the z_i will be ordered according to siteid. ie $z_i <_e z_j \Leftrightarrow z_i.s < z_j.s$. Clearly O must be inserted in order to maintain the ordering by site identifier. This uniquely defines the insertion position amongst the z_i.

However more generally there can be causal orderings amongst the z_i, and this complicates the problem of where to correctly insert O amongst the z_i.

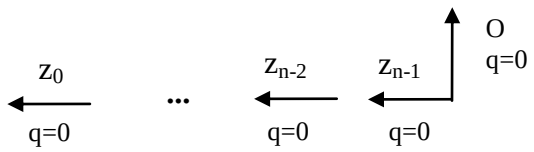
Claim: $z_j \rightarrow z_i \Rightarrow z_i <_e z_j$
Proof: $z_j \rightarrow z_i$ $\Rightarrow$ z_i was executed in the context of z_j
$\Rightarrow$ z_i dominates z_j
$\Leftrightarrow$ $z_i <_e z_j$

Proposal: Restructure this explanation as follows

1. The case of an insertion of an interval where all intervals in the repository are in $\chi(v)$ leads to a left to right scan
2. The case of an insertion of an interval where all intervals in the repository are not in $\chi(v)$ leads to a right to left to scan
3. Explain how to use above two solutions to solve the general case.

Case where all intervals in the repository are not in $\chi(v)$
Let the repository consist of intervals $z_0,z_2,...z_{n-1}$ where each $z_i \notin \chi(v)$ and the z_i are ordered according to the effects relation $<_e$.

The insertion position of a given operation can be achieved with a right to left scan of the z_i. The repository $[z_0,...,z_{n-1}]$ can be considered a sequence of contextually serialised sequence of operations in reverse order $L = z_{n-1},...,z_0$ that all insert at q = 0. This totally ordered sequence is compatible with the partial ordering defined by the precedes relation because $z_j \rightarrow z_i \Rightarrow z_i <_e z_j$.



We can assume O also inserts a character at q = 0. Also O || L so it makes sense to consider IT(O,L) and IT(L,O). We know we simply IT O against z_{n-1} then z_{n-2} and so on. Furthermore all the insertions are at the same q-position so therefore we compare site ids and only increment the q-position of the loser. Once O has lost (so O.q becomes nonzero) O will continue to lose against the remaining z_i.

As a result this shows that the correct insertion position of O is determined by a right to left scan and comparison of site identifiers.

Let Z[i] be the concurrent operations ordered by the effects relation, indexed from 0 onwards. Then the following algorithm is used to insert operation O at the appropriate position:

```
// Start at the last Z[i]
int i = |Z|-1;

// Scan from right to left while O dominates Z[i] according to site identifiers
// Yield i = -1 if O dominates every Z[i].
while(i >= 0 && O.s < Z[i].s) --i;

// Insert O at position i+1
Z.insert(i+1, O);
```

References

[1]	David Barrett-Lennard. Vector Time. Feb 2008