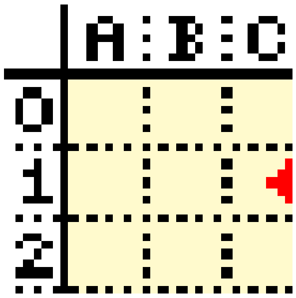
# Operation Transforms for a Distributed Shared Spreadsheet

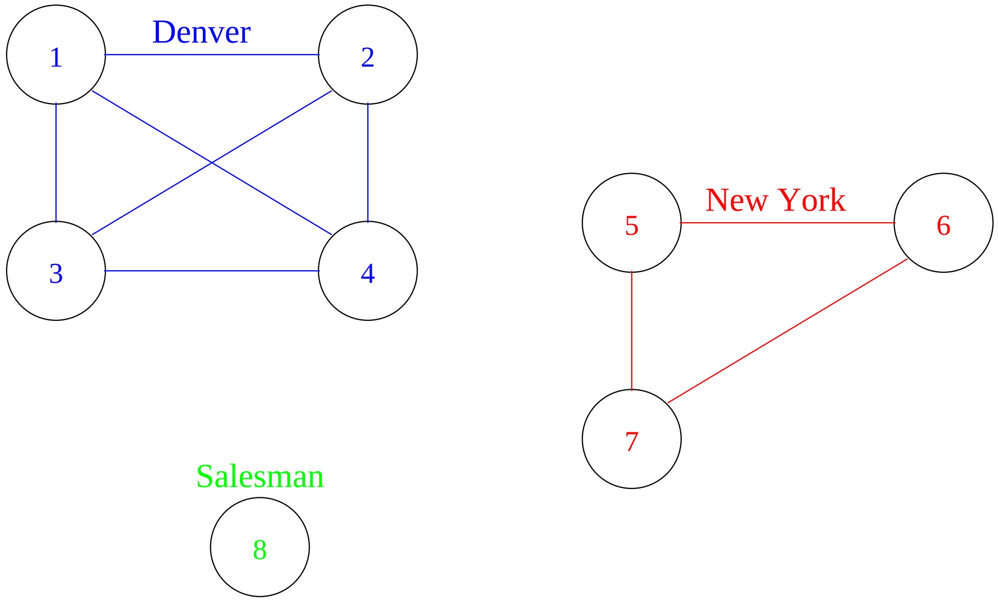



Christopher R. Palmer
School of Computer Science
Carnegie-Mellon University
crpalmer+@cs.cmu.edu

Gordon V. Cormack
Department of Computer Science
University of Waterloo
gvcormack@plg.uwaterloo.ca

# Goals and Motivation

- Multiple users sharing the same spreadsheet.
- Low bandwidth and part-time connections.
- Salesman with a laptop and a WAN connection.




- Problem #1: Consistency (when in a stable state).

# Basic Consistency

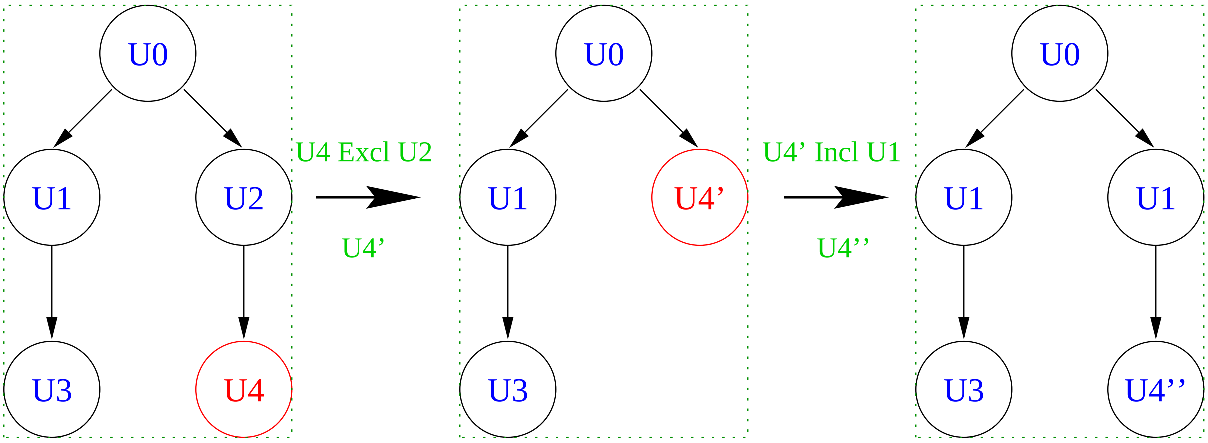
- No network $\Rightarrow$ no global semaphores and no centralized consistency algorithm.
- That leaves: Act and Inform.
- Users need a causal order and we can create a total order.
- If updates are applied in total order, consistency.
- If updates are invertible, this is possible.
- Example: updates $u_1 < u_2 < \cdots < u_6$:

| | | |
|---|---|---|
| Executed: | $u_1\ u_2\ u_3\ u_5\ u_6$ | Receive $u_4$: undo $u_6$ then $u_5$ |
| Executed: | $u_1\ u_2\ u_3$ | Can now do $u_4$ and redo $u_5/u_6$ |
| Executed: | $u_1\ u_2\ u_3\ u_4\ u_5\ u_6$ | |

- But does the consistent state make sense?

# Transforms with Undo/Redo

- Operation transforms can be used to fix updates.
- Transform operations given the state of your world so that they make sense (Sun's algorithm).
- Two Transforms: include or exclude to/from a context.

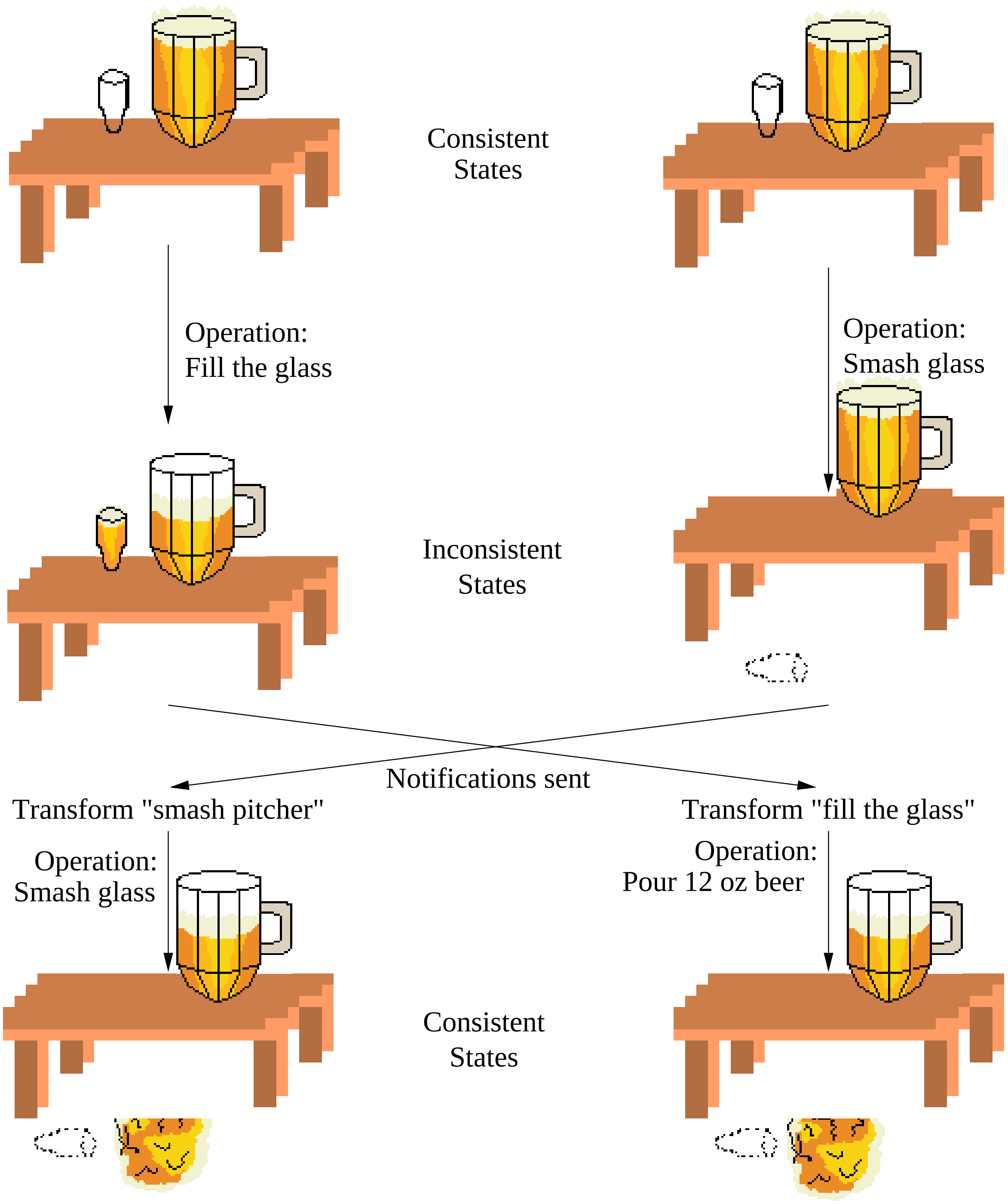



- Total order is good for defining meaning, but inverses can be restrictive:
  - Inverses may not exist (eg: real world interaction).
  - Inverses may be very expensive.
    - ∗ Eg: deleting 1,000 rows with 1,000 columns from a spreadsheet.

  ⇒ every site must remember 1,000,000 cells!

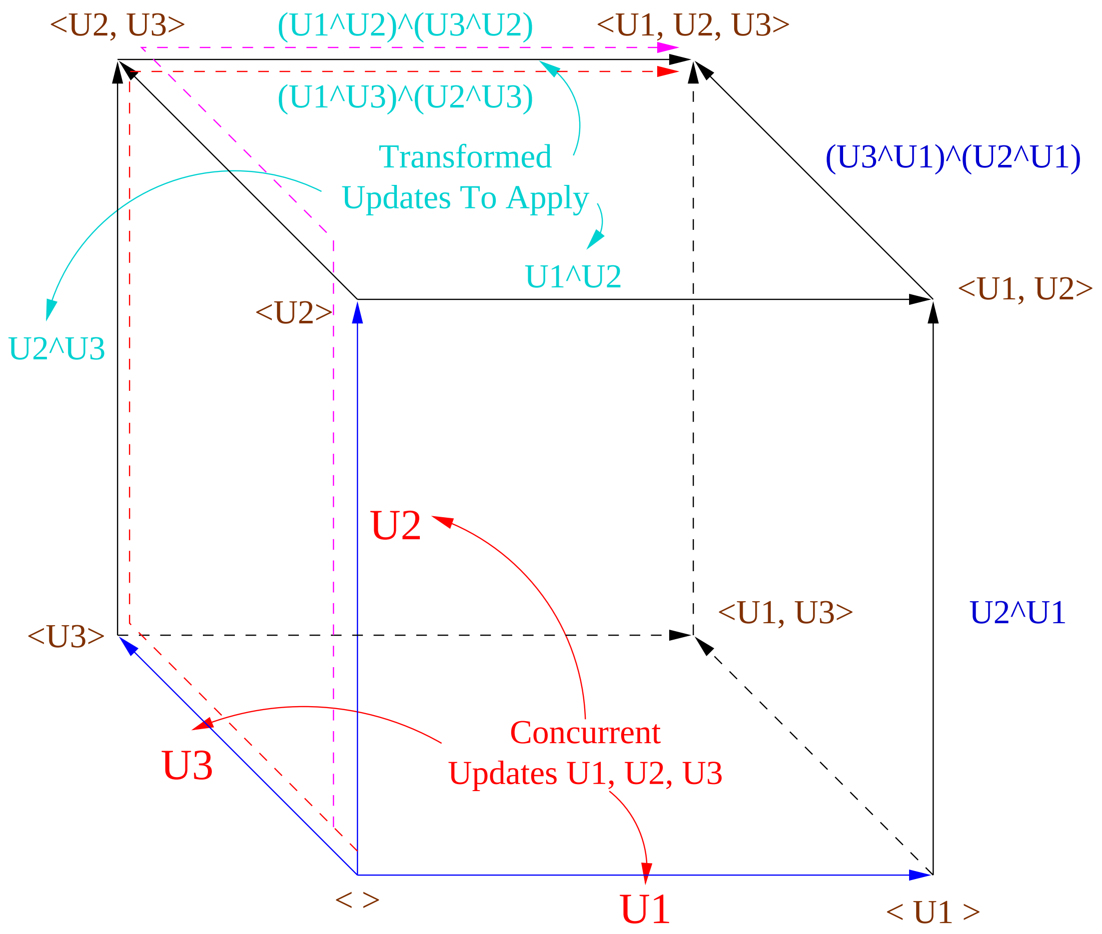
  ⇒ every undo/redo affects 1,000,000 cells!

# Transforms without Inverses I



# Transforms without Inverses II

- Contingency graph: vertices are the meaningful states, edges define updates that transform between states.




- Forward looking transforms are good (no inverses!).
- Lack of total order $\Rightarrow$ all paths equal.
- Forward Looking + Total Order $\Leftrightarrow$ Cormack's CCU.

# Calculus for Concurrent Update

- Define a total and partial order for vector timestamps:

$$
\begin{array}{l}
(x_1, \cdots, x_n) \text{ happens before } (y_1, \cdots, y_n) \\
\qquad \text{if } \; x_i \le y_i \text{ for all } i \text{ and } x_j < y_j \text{ for some } j
\end{array}
$$

$$
\begin{array}{l}
(x_1, \cdots, x_n) < (y_1, \cdots, y_n) \\
\qquad \text{if } \; x_1 < y_1 \text{ or} \\
\qquad \qquad x_1 = y_1 \text{ and } (x_2, \cdots, x_n) < (y_2, \cdots, y_n)
\end{array}
$$

- Transform #1: / ("after"): $f/g$ means the same thing as $f$ given that a concurrent $g$ intervened earlier in the total order.

- Transform #2: \ ("before") dual of / (defined by TP1) and is not meaning preserving.

$$f/g(g(X)) = g\backslash f(f(X)) \text{ for any state} X \qquad \text{TP1}$$

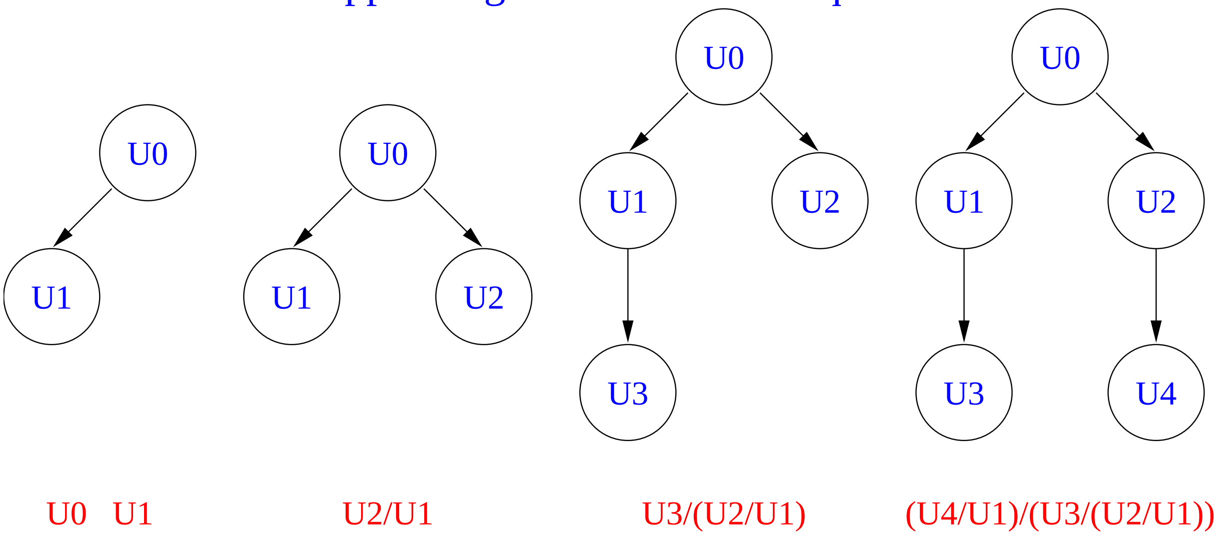
$$(h/g)/(f\backslash g) = (h/f)/(g/f) \qquad \text{TP1.5}$$

- TP1 allows us to reorder pairs of updates.

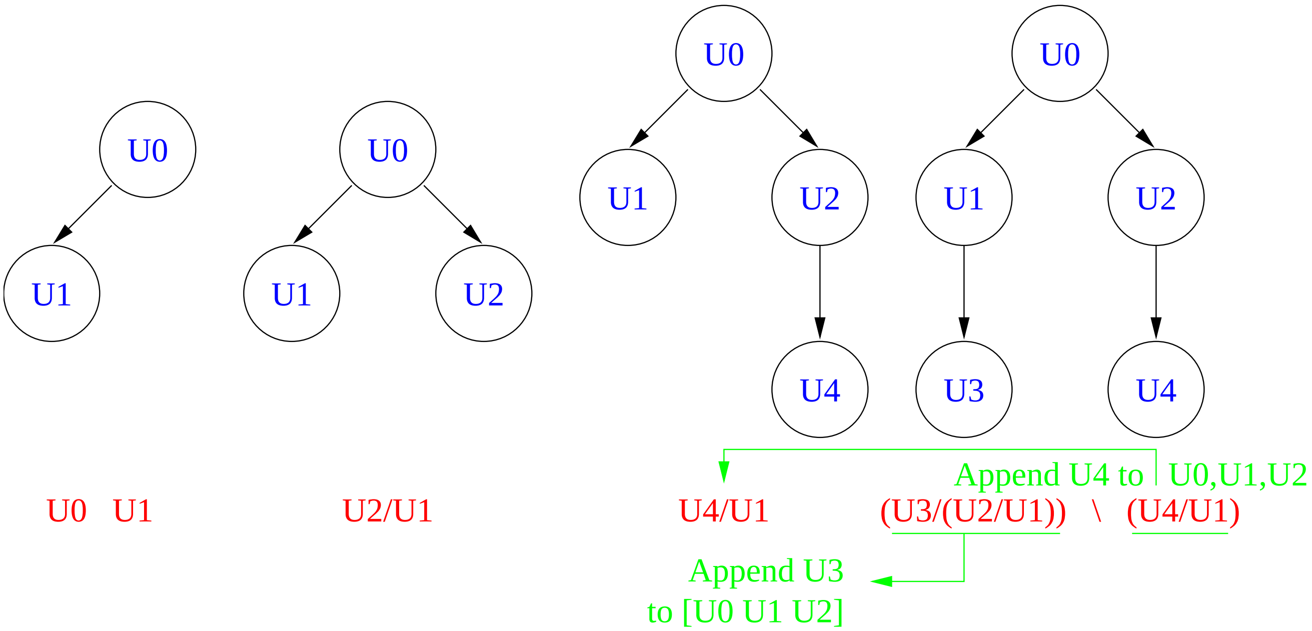
- TP1.5 allows us to append to a canonical update sequence.

# CCU Algorithm Overview

## Appending to a Canonical Sequence




## Inserting Into a Canonical Sequence



# Spreadsheet Model (I)

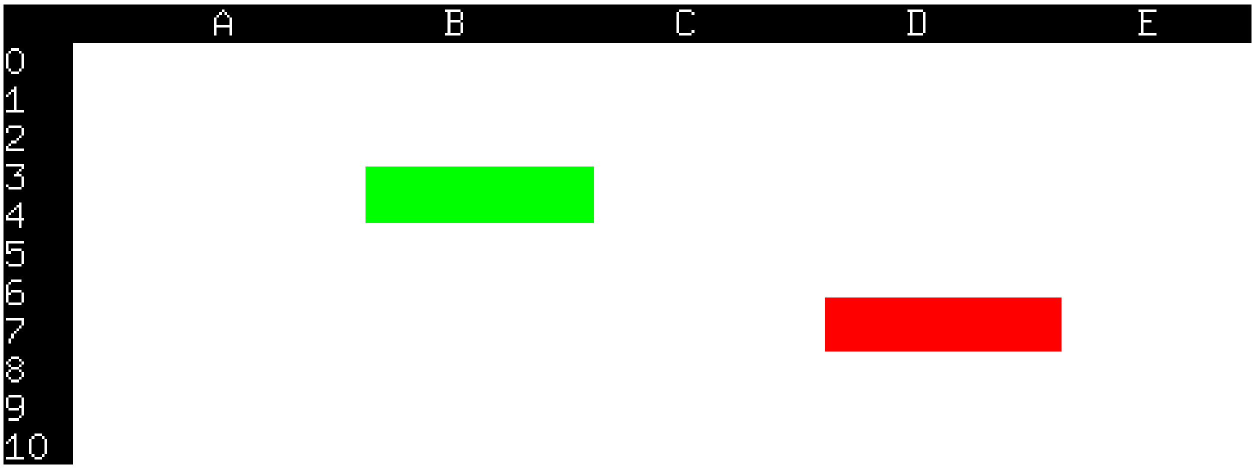
| | A | B | C | D |
|---|---|---|---|---|
| 0 | [0,0] | [0,1] | [0,2] | [0,3] |
| 1 | [1,0] | [1,1] | [1,2] | [1,3] |
| 2 | [2,0] | [2,1] | [2,2] | [2,3] |
| 3 | [3,0] | [3,1] | [3,2] | [3,3] |
| 4 | [4,0] | [4,1] | [4,2] | [4,3] |

- A spreadsheet is a table that holds numbers, text and formulas.
- Entries in the table are cells.
- Formulas can reference fixed or relative cell positions (copy).
- Formulas can reference rectangular ranges of cells. Ranges of cells can be named.
- Many functions update the spreadsheet, eg: `copy`, `move`, `increment`.
- Sufficient set of update operations:
  - `set([row, col], value)` – store a value in the spreadsheet.
  - `format(col, code)` – set formatting information.
  - `insert(row, nrows, col, ncols)` – insert `nrows` rows before `row` and `ncols` columns before `col`.
  - `delete(row, nrows, col, ncols)` – delete `nrows` rows beginning with row `row` and `ncols` columns beginning with `col`.
  - `define(name, [row1, col1], [row2, col2])` – add a definition binding `name` to the range with rows `row1..row2` and columns `col1..col2`.
  - `undefine(name)` – removing any name binding for `name`.

# Spreadsheet Model (II)

```
Store:  C : 2 dimensional array of cells
        F : 1 dimensional array of column formatting codes
        M : associative array M[name] holds the pair of
            cells defining the rectangular range defined
            for name.

        C has dimensions maxr x maxl
        F has maxc entries

set([row, column], value) is
   C[row, column] = sfix(value)

format(column, formatcode) is
   F[column] = formatcode

delete(row1, nrows, col1, ncols) is
   FOR all 1 <= c <= maxc and 1 <= r <= maxr
       C[r, c] = dfix(C[r,c])
   C[row1..maxr-nrows, 1..maxc]=C[row1+nrows..maxr, 1..maxc]
   C[1..maxr, col1..maxc-ncols]=C[1..maxr, col1+ncols..maxc]
   F[col1..maxc-ncols]         =F[col1+ncols..maxc]

insert(row1, nrows, col1, ncols)
   FOR all 1 <= c <= maxc and 1 <= r <= maxr
       C[r, c] = dfix(C[r,c])
   C[row1+nrows..maxr,1..maxc]=C[row1..maxr-nrows,1..maxc]
   C[1..maxr,col1+ncols..maxc]=C[col1..maxc-ncols,1..maxc]
   F[col1+ncols..maxc]        =F[col1..maxc-ncols]
   C[row1..row1+nrows-1, 1..maxc] = empty
   C[1..maxr, col1..col1+ncols-1] = empty
   F[col1..col1+ncols-1]          = defaultformat

define(name, [row1, col1], [row2, col2]) is
   M[name] = <[row1, col1], [row2, col2]>

undefine(name) is
   M[name] = nil
```

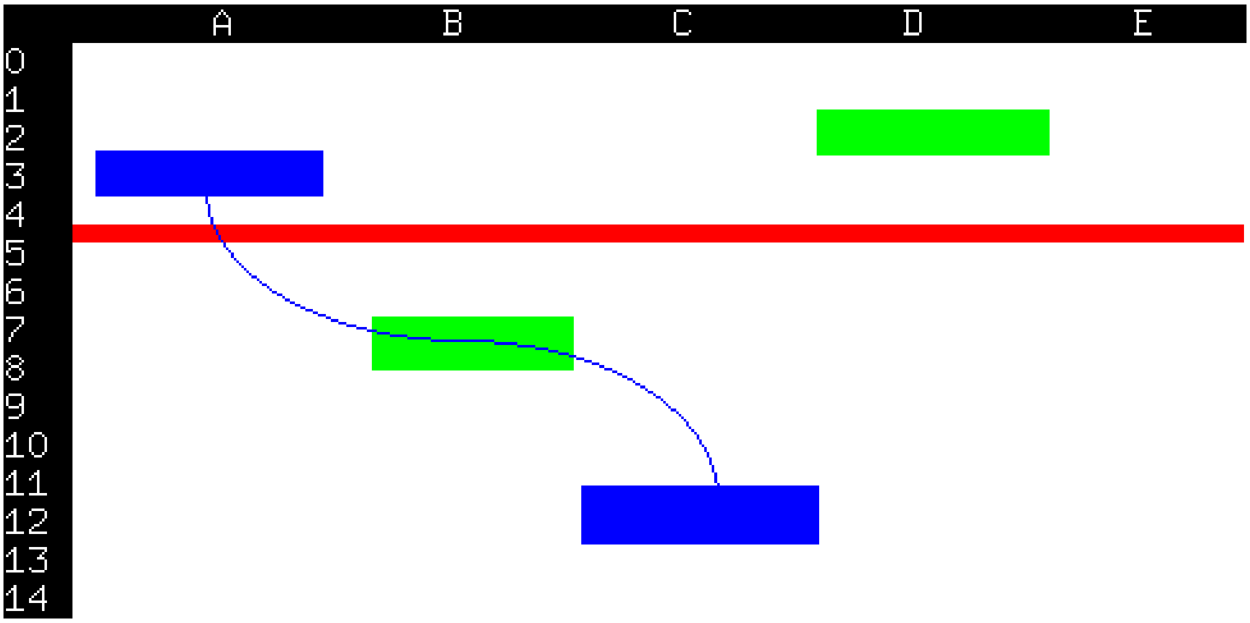
# Transforming `set` and `set`




- Red and green areas show different `set` cells.
- If different, apply both. If the same, pick one to apply.
- Arbitrary choice: keep the last `set` (pretend they are serialized).

```
set([r1,c1], v1) / set([r2,c2]), v2) = set([r1,c1], v1)

set([r1,c1], v1] \ set([r2,c2], v2) =
    set([r1,c1], v1)     ([r1,c1] != [r2,c2])
    ident                ([r1,c1] = [r2,c2])
```

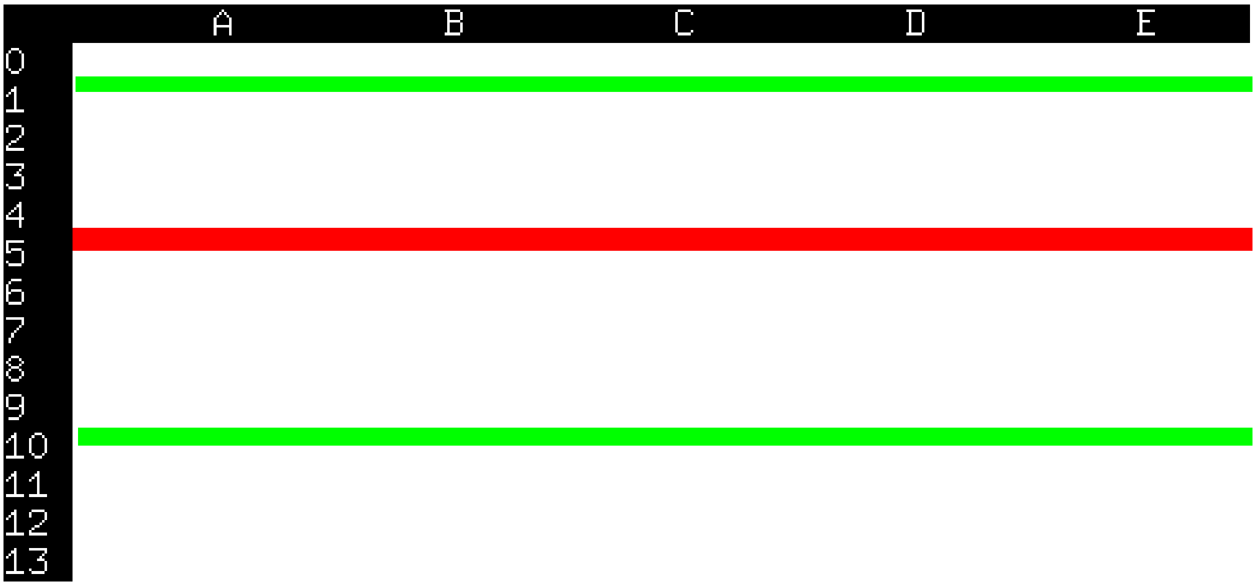
# Transforming `set` and `insert`




- Red line shows where we `insert` (before row 5).
- Green cells show different cases for the `set` cell.
- Must correct cell being `set` and the formula (blue cells).

```
insert(r1,nr,c1,nc) ^ set([r2,c2], v2) = insert(r1, nr, c1, nc)

set([r1,c1], v1) ^ insert(r2,nr,c2,nc) =
   set[rf(r1), cf(c1), cf(rf(v1)))
where rf(literal) = literal
      rf(@f(e1, ..., en)) = @f(rf(e1), ..., rf(en))
      rf([rr1,rc1]..[rr2,rc2]) = [rf(rr1),rc1]..[rf(rr2),rc2]
      rf([rr1,cc1]) = [rr1,cc1]}             (rr1 >= r2)
      rf([rr1,rc1]) = [rr1+nr,rc1]}          (rr1 < r2)
```

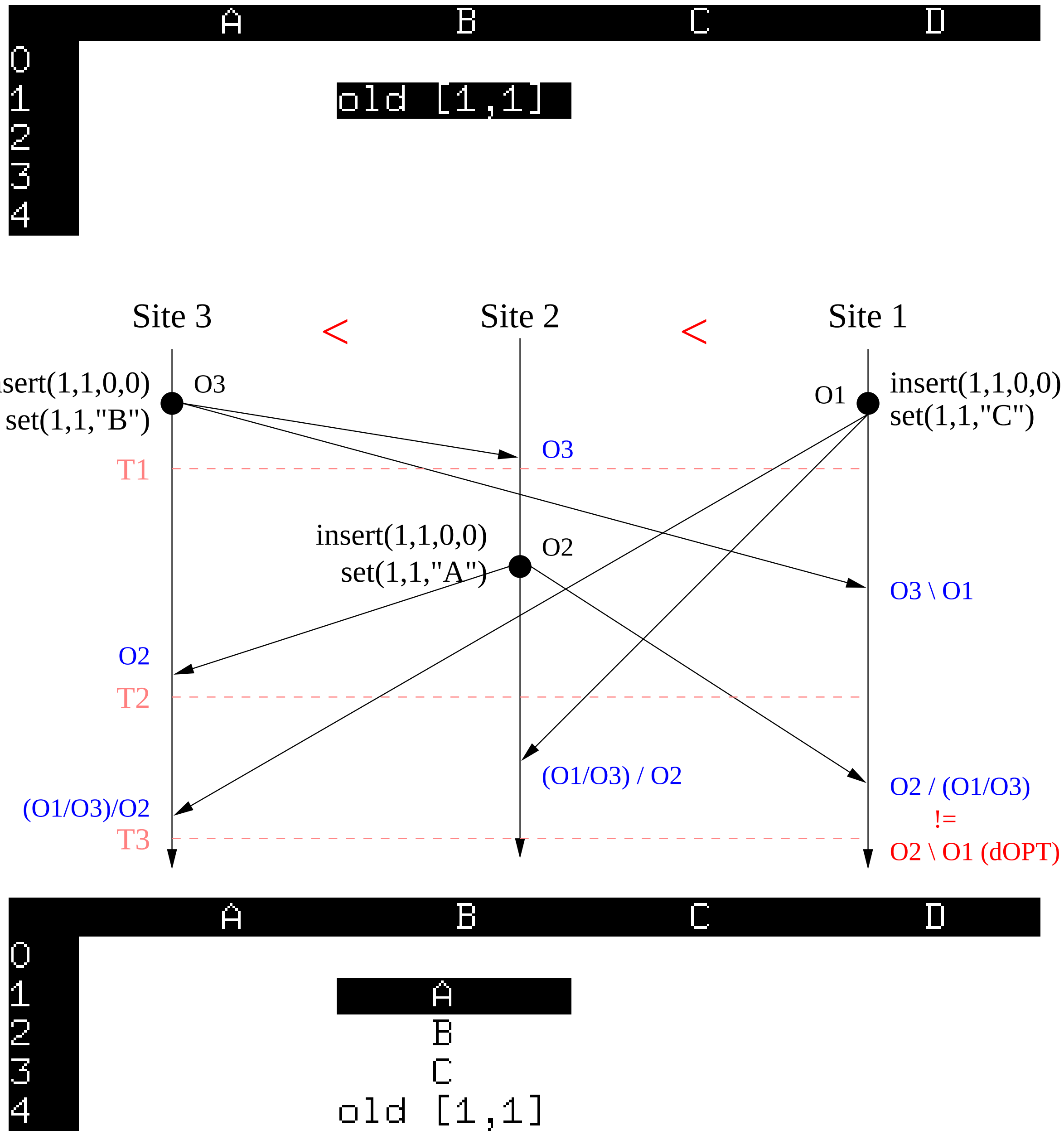
# Transforming `insert` and `insert`



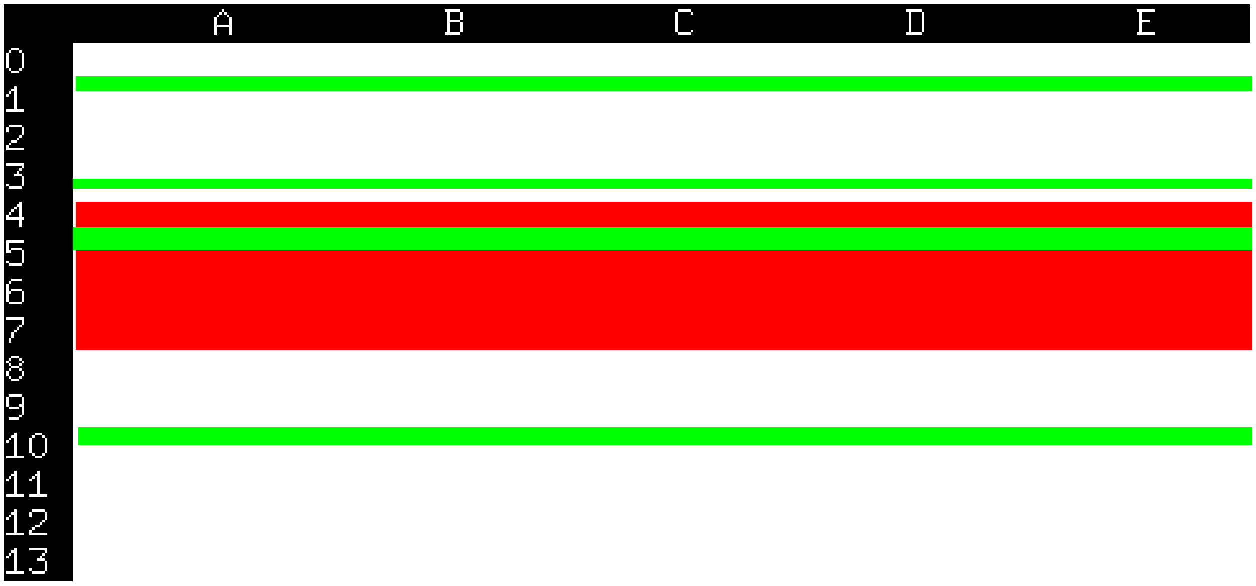

- Green lines show a pair of `inserts`.
- Red line represents two concurrent inserts at the same point.
- Must offset row references and use total order to offset only one of the two updates at the red point.

```
insert(r1,nr1,?,?) / insert(r2,nr2,?,?) =
   insert(r1,nr1,?,?)           (r1 <  r2)
   insert(r1+nr2,nr1,?,?)       (r1 >= r2)

insert(r1,nr1,c1,nc1) \ insert(r2,nr2,c2,nr2) =
   insert(r1,nr1,?,?)           (r1 <= r2)
   insert(r1+nr2,nr1,?,?)       (r1  > r2)
```

# The dOPT Puzzle



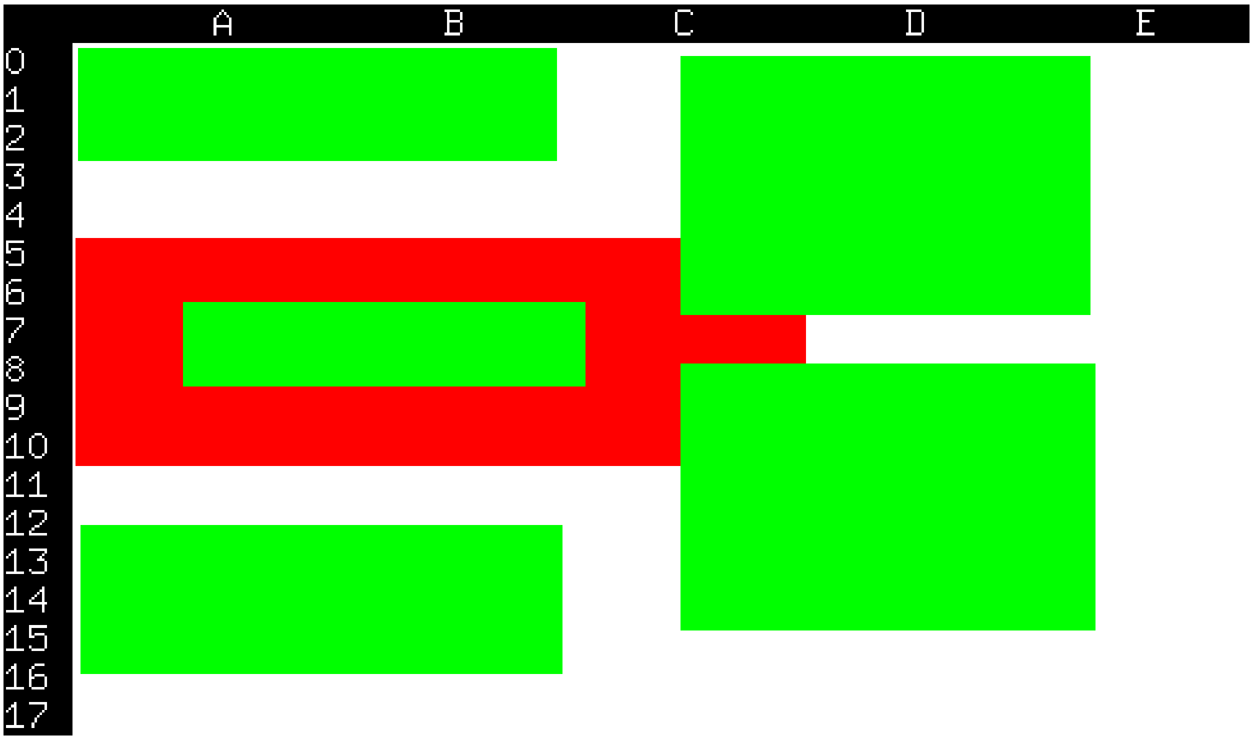
# Transforming `delete` and `insert`




- Green lines show different `insert` points. Before row 4 is a special case.
- Red area a `delete` range.
- Have to correct `insert` point, `delete` point and `delete` range.
- The `delete` must be translated if the `insert` comes before it and the range extended if the `insert` is subsumed.

```
delete(r1,nr1,c1,nc1) ^ insert(r2,nr2,c2,nc2) =
   delete(rf(r1), rf(r1+nr1)-rf(r1), cf(c1), cf(c1+nc1))
where rf(r) = r                   (r < r2)
      rf(r) = r+nr2               (r >= r2)

insert(r1,nr1,?,?) ^ delete(r2,nr2,?,?) =
   insert(r1,nr1,?,?)         (r1 < r2)
   ident                      (r2 <= r1 < r2+nr2)
   insert(r1-nr2,nr1,?,?)     (r1 >= r2+nr2)
```

# Transforming `delete` and `delete`

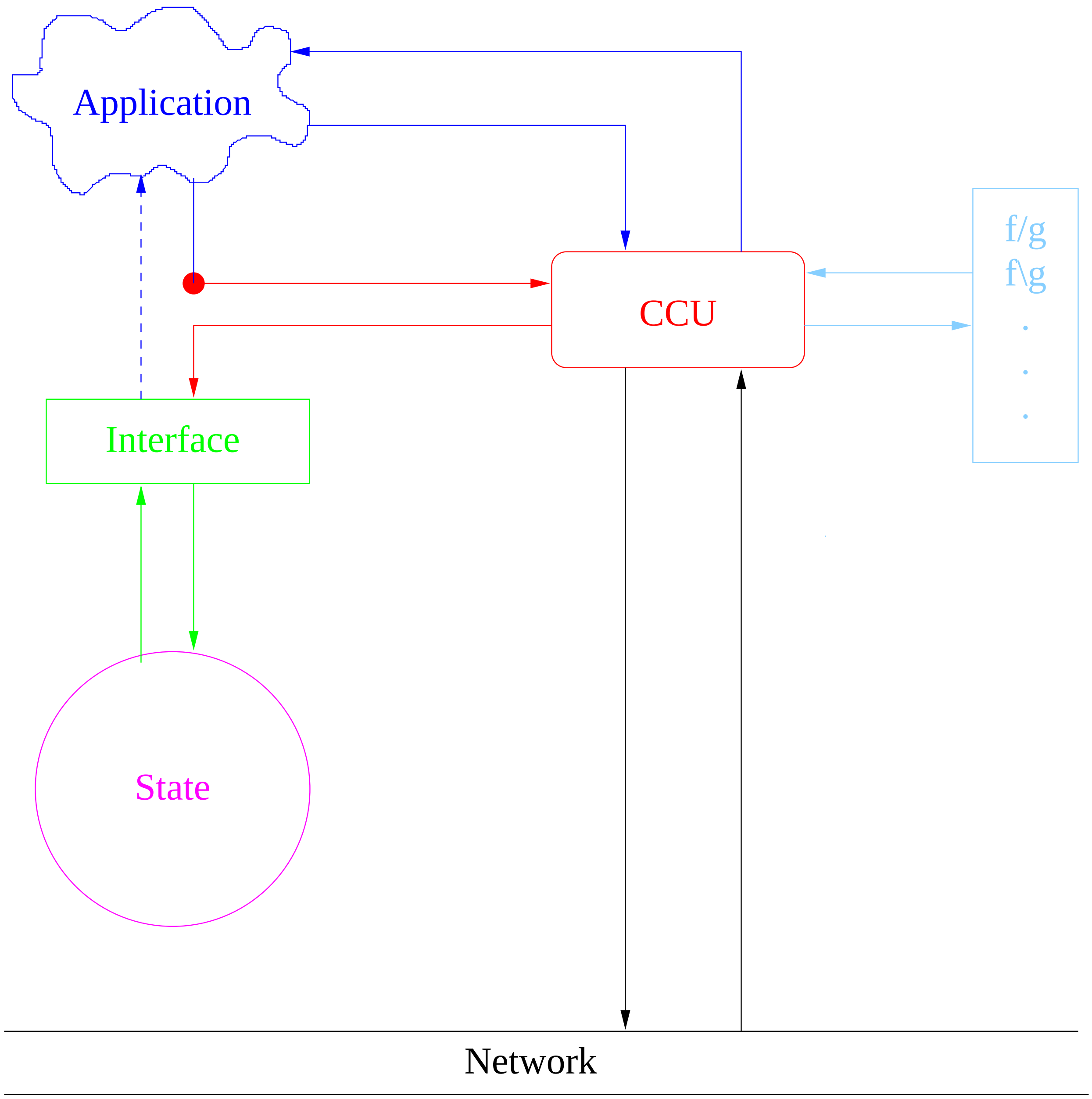



- Red and green areas show possible interactions between `delete`s.

- Again must translate deletion points and correct ranges.

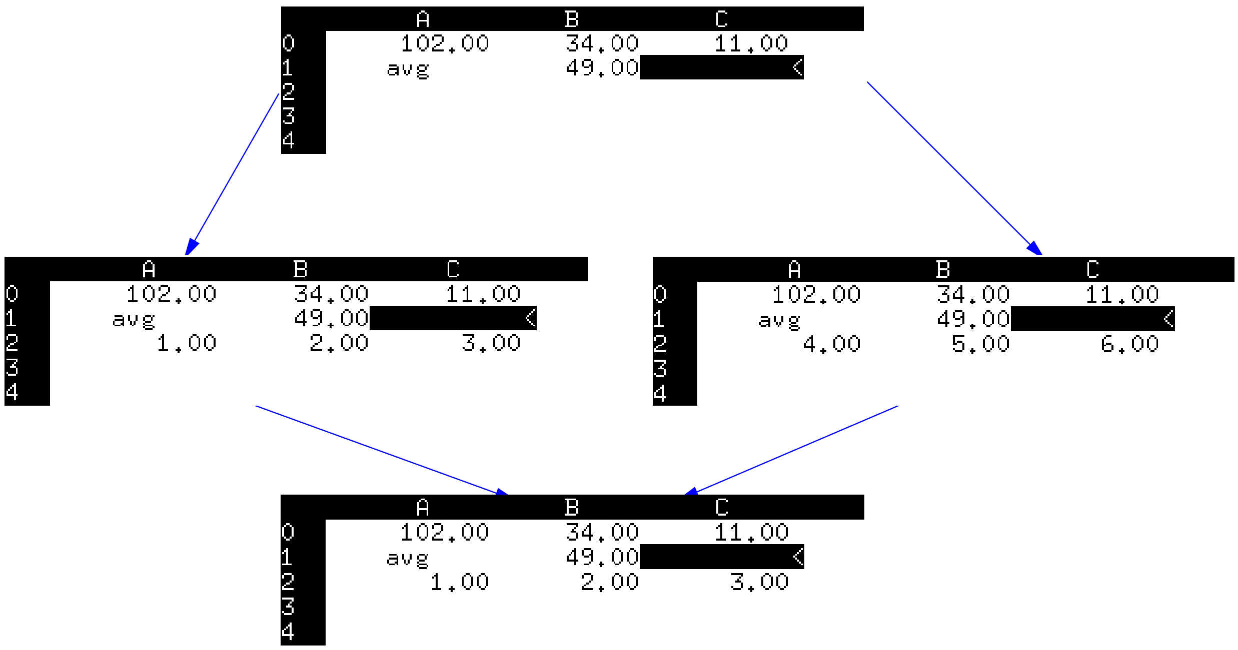
```
delete(r1,nr1,c1,nc1) ^ delete(r2,nr2,c2,nc2) =
   delete(fr(r1),fr(r1+nr1)-fr(r1),fc(c1),fc(c1+nc1)-fc(c1))
where
   fr(r) = r                (r < r2)
   fr(r) = r2               (r2 <= r <= r2+nr2)
   fr(r) = r - nr2          (r > r2+nr2)
```

# Implementation Approach



# Implementation Experiences

- `copy` operation is not needed (it is in proceedings).
- Can model `copy` by transforming the formula and then doing a `set`.
- Definitions are correct, but there can still be surprises:




- Easy to fix, `insert` a row/column when spreadsheet extended.
- Moral: transforms must be actually used to evaluate their utility.

# Conclusions

- CCU has a total order (easier to define meaning).
- Total order can give trivial atomic sequences of updates.
- CCU doesn't use inverse operations for consistency.
- CCU allows us to express larger concurrent systems.
- Distributed asynchronous concurrency control algorithms can be applied to larger problems.