4A. Assume the schemas

Local0(Id, Loc)
Op1(Id, In, Loc)
Op2(Id, In1, In2, Loc)
Local(Id, Loc, Num)

Ip is $\pi_{Id, Loc, Num=1}(Local0) \cup$

$\pi_{Id, Loc, Num = Num+1}(Local \bowtie Op1) \cup$

$\pi_{Id, Loc, Num=Num1+Num2+1}(\rho_{In \to In1, Num \to Num1}(Local) \bowtie Op2 \bowtie \rho_{In \to In2, Num \to Num2}(Local))$

4Ba. We can be slightly less restrictive than the definition in the paper. Rather than requiring all variables in a computed predicate to be in a base predicate, we only need variables in an allowed binding pattern to be in a base predicate*. For example

Cost(Item, C) :- Price(Item, C1), Tax(Item, C2), Sum(C1, C2, C).

Note that C does not appear in ~~the~~ any base predicate, so this rule does not satisfy "strongly safe" from the paper. But it does satisfy the condition above, because the first two positions of Sum are variables that appear in base predicates.

[Note: This definition could be made even less restrictive, to allow bindings from "safe" occurrences of computed preds.

E.g. DiscountCost(Item, D) :- Price(Item, C1), Tax(Item, C2), Sum(C1, C2, C), Prod(C, .90, D).]

this Sum is "safe" because C1, C2 in base preds

this literal is OK since C comes from a "safe" occurrence of a computed pred.

* or have constants

4Ba cont

Because each computed predicate is only used with a finite combination of values (since those values come from the base, which is finite), and each one yields only finitely many new values for each binding, there are only finitely many new constants introduced. Hence there is only a finite combination of values for any predicate to be true.

4Bb

Local(Id, Loc, K) :- Op1(Id, In, Loc), Local(In, Loc, M), Sum(M, 1, K).

Does not satisfy the condition, since one of M or K would need to be in a base predicate to do so.
The 3rd rule has similar violations.

4Bc. We can add an extensional predicate with allowed values, and use it to restrict variables in computed predicates:

Value(1). Value(2). Value(3). ... Value(100).

Rules 2 and 3 become

Local(Id, Loc, K) :- Op1(Id, In, Loc), Local(In, Loc, M), Value(M), Sum(M, 1, K).

Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc), Local(In1, Loc, M), Local(In2, Loc, N), Value(M), Value(N), Sum(M, N, J), Value(J), Sum(J, 1, K).

#II [Note: Some people misread the term "range restricted" from the paper. It only means the variable appears somewhere in the body of the rule.]

4Ca.

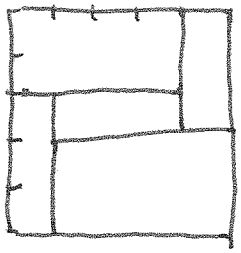
Fill(2,4) :- Tile(2,4).
Fill(1,3) :- Tile(1,3).
Fill(3,1) :- Fill(1,3).
Fill(3,2) :- Fill(3,1), Fill(3,1), Sum(1,1,2), LessEq(3,10), LessEq(2,10).
Fill(2,3) :- Fill(3,2).
Fill(2,7) :- Fill(2,3), Fill(2,4), Sum(3,4,7), LessEq(2,10), LessEq(7,10).

4Cb. As noted in class, the Fill rules only work with rectangular decompositions.

Thus to get Fill(5,5) we need either Fill(5,1) or Fill(5,2). (We might also need Fill(5,3) or Fill(5,4), but those require the other 2.)

- We can't get Fill(5,1). The only tile we can use here is Tile(1,3), which leaves Fill(2,1), which is smaller than any tile.
- To get Fill(5,2), we could use Fill(5,1) twice, but we can't do Fill(5,1). We can try Fill(2,5), which would require Fill(2,1) or Fill(2,2). However, neither tile fits inside either of those areas.

4Cc.



4Da. The program isn't equivalent. We can flip tiles, but we can't flip filled areas. For example, in the previous problem, if we can't use

Fill(2,3) :- Fill(3,2)

we are unable to derive Fill(4,7).

4D6. These two programs are equivalent.

- It's pretty easy to see that the new program can't derive anything the first one couldn't.

Suppose we use the first ~~rule~~ new rule:

Fill(i,j) :- Tile(j,i)

Then we could have used the following two instances of rules in the original program.

~~Fill(i,j) :- T~~ Fill(j,i) :- Tile(j,i).

Fill(i,j) :- Fill(j,i).

If we use an instance of the second new rule

Fill(i,j) :- Fill(i2,j), Fill(i2,j), Sum(i1,i2,i),
LessEq(i,10), LessEq(j,10).

Then we could get the same effect with instances of orginal rules:

Fill(j,i1) :- Fill(i1,j).

Fill(j,i2) : Fill(i2,j).

Fill(j,i) :- Fill(j,i1), Fill(j,i2), Summ(i1,i2,i),
LessEq(j,10), LeoEq(i,10).

Fill(i,j) :- Fill(j,i).

- The more difficult part is to show that the new program derives everything the first program can. Suppose ~~th~~ for a given database d there is a fact we can derive with the original program that the new program can't derive. Let Fill(i,j) be such a fact that is minimal: for any other non-derivable fact Fill(i',j') we have $i' \geq i$ or $j' \geq j$.

4Db. (cont.)

Consider 3 cases, corresponding to which rule derived Fill(i,j).

Case 1 Fill(M,N) :- Tile(M,N). This rule is in the new program, so we can derive Fill(i,j)

Case 3 Fill(i,j) was derived with rule 3, from Fill(i,j1) and Fill(i,j2). These last two facts are "smaller" than Fill(i,j), so by the minimality of Fill(i,j), they are derivable in the new program. So we can use this same rule in the new program to derive Fill(i,j)

Case 2 Fill(i,j) was derived from Fill(j,i) with rule 2.

Case 2.1 Fill(j,i) was derived by rule 1 from Tile(j,i).

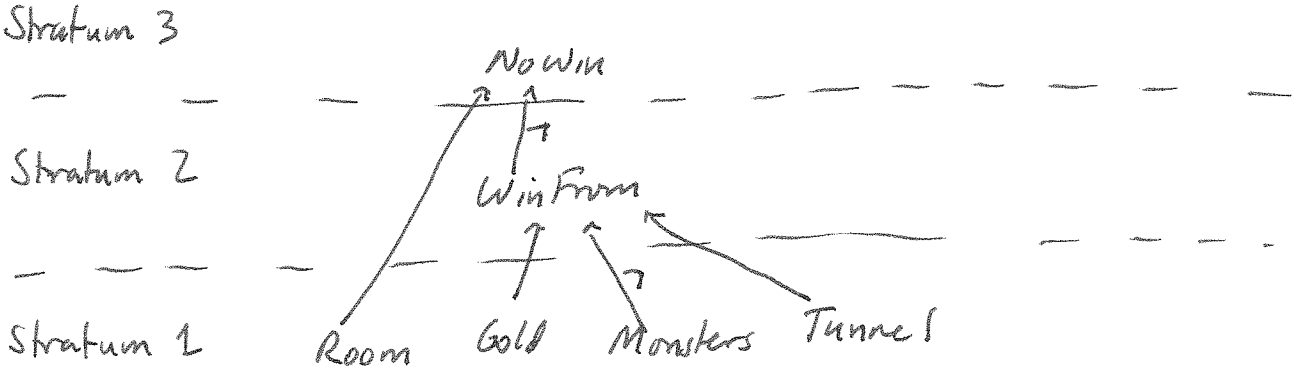
Then we can use Fill(i,j) :- Tile(j,i) in the new program to derive Fill(i,j).

Case 2.3 Fill(j,i) was derived by rule 1 from Fill(j,i1) and Fill(j,i2). It must also be the case that Fill(i1,j) and Fill(i2,j) are derivable in the original program. Since these are smaller than Fill(i,j), they are derivable in the new program. Thus we can use the second new rule to derive Fill(i1+i2,j) = Fill(i,j) from Fill(i1,j) and Fill(i2,j).

So we can always derive Fill(i,j) in the new program.

Note: If your argument didn't address the use of the new rule Fill(M,N) :- Tile(N,M), then it is incomplete, because we don't have equivalence without this rule.

4Eq. There are several possible stratifications. One is



4Eb.

Stratum 1 has the extensional DB.

Stratum 2 adds

WinFrom(treasury).
WinFrom(batcave).
WinFrom(lair).

Stratum 3 adds

NoWin(den).
NoWin(armory).
NoWin(prison).