Solver.h line 239:

```
inline uint32_t Solver::abstractLevel (Var x) const   { return 1 << (level[x] & 31); }
```

Solver.C line 204

```
/*_________________________________________________________________________________________________
|
|  analyze : (confl : Clause*) (out_learnt : vec<Lit>&) (out_btlevel : int&)  ->  [void]
|
|  Description:
|    Analyze conflict and produce a reason clause.
|
|    Pre-conditions:
|      * 'out_learnt' is assumed to be cleared.
|      * Current decision level must be greater than root level.
|
|    Post-conditions:
|      * 'out_learnt[0]' is the asserting literal at level 'out_btlevel'.
|
|  Effect:
|    Will undo part of the trail, upto but not beyond the assumption of the current decision
 level.
|________________________________________________________________________________________________@*/
void Solver::analyze(Clause* confl, vec<Lit>& out_learnt, int& out_btlevel)
{
    int pathC = 0;
    Lit p     = lit_Undef;

    // Generate conflict clause:
    //
    out_learnt.push();      // (leave room for the asserting literal)
    int index   = trail.size() - 1;
    out_btlevel = 0;

    do{
        assert(confl != NULL);          // (otherwise should be UIP)
        Clause& c = *confl;

        if (c.learnt())
            claBumpActivity(c);

        for (int j = (p == lit_Undef) ? 0 : 1; j < c.size(); j++){
            Lit q = c[j];

            if (!seen[var(q)] && level[var(q)] > 0){
                varBumpActivity(var(q));
                seen[var(q)] = 1;
                if (level[var(q)] >= decisionLevel())
                    pathC++;
                else{
                    out_learnt.push(q);
                    if (level[var(q)] > out_btlevel)
                        out_btlevel = level[var(q)];
                }
            }
        }

        // Select next clause to look at:
        while (!seen[var(trail[index--])]);
        p     = trail[index+1];
        confl = reason[var(p)];
```

```
        seen[var(p)] = 0;
        pathC--;

    }while (pathC > 0);
    out_learnt[0] = ~p;
```

Solver.C line 265

```
    // Simplify conflict clause:
    //
    int i, j;
    if (expensive_ccmin){
        uint32_t abstract_level = 0;
        for (i = 1; i < out_learnt.size(); i++)
            abstract_level |= abstractLevel(var(out_learnt[i])); // (maintain an abstraction
 of levels involved in conflict)

        out_learnt.copyTo(analyze_toclear);
        for (i = j = 1; i < out_learnt.size(); i++)
            if (reason[var(out_learnt[i])] == NULL || !litRedundant(out_learnt[i], abstract_
level))
                out_learnt[j++] = out_learnt[i];
    }else{
        out_learnt.copyTo(analyze_toclear);
        for (i = j = 1; i < out_learnt.size(); i++){
            Clause& c = *reason[var(out_learnt[i])];
            for (int k = 1; k < c.size(); k++)
                if (!seen[var(c[k])] && level[var(c[k])] > 0){
                    out_learnt[j++] = out_learnt[i];
                    break; }
        }
    }
    max_literals += out_learnt.size();
    out_learnt.shrink(i - j);
    tot_literals += out_learnt.size();

    // Find correct backtrack level:
    //
    if (out_learnt.size() == 1)
        out_btlevel = 0;
    else{
        int max_i = 1;
        for (int i = 2; i < out_learnt.size(); i++)
            if (level[var(out_learnt[i])] > level[var(out_learnt[max_i])])
                max_i = i;
        Lit p             = out_learnt[max_i];
        out_learnt[max_i] = out_learnt[1];
        out_learnt[1]     = p;
        out_btlevel       = level[var(p)];
    }


    for (int j = 0; j < analyze_toclear.size(); j++) seen[var(analyze_toclear[j])] = 0;    /
/ ('seen[]' is now cleared)
}


// Check if 'p' can be removed. 'abstract_levels' is used to abort early if the
algorithm is
// visiting literals at levels that cannot be removed later.
bool Solver::litRedundant(Lit p, uint32_t abstract_levels)
{
    analyze_stack.clear(); analyze_stack.push(p);
    int top = analyze_toclear.size();
    while (analyze_stack.size() > 0){
```

```
        assert(reason[var(analyze_stack.last())] != NULL);
        Clause& c = *reason[var(analyze_stack.last())]; analyze_stack.pop();

        for (int i = 1; i < c.size(); i++){
            Lit p  = c[i];
            if (!seen[var(p)] && level[var(p)] > 0){
                if (reason[var(p)] != NULL && (abstractLevel(var(p)) & abstract_levels) != 0
){
                    seen[var(p)] = 1;
                    analyze_stack.push(p);
                    analyze_toclear.push(p);
                }else{
                    for (int j = top; j < analyze_toclear.size(); j++)
                        seen[var(analyze_toclear[j])] = 0;
                    analyze_toclear.shrink(analyze_toclear.size() - top);
                    return false;
                }
            }
        }
    }

    return true;
}
```