January 29, 2026 at 12:33

**1. Intro.** This program is part of a series of "SAT-solvers" that I'm putting together for my own education as I prepare to write Section 7.2.2.2 of *The Art of Computer Programming*. My intent is to have a variety of compatible programs on which I can run experiments to learn how different approaches work in practice.

The other programs in the series solve instances of SAT, but this one is different: It's a *preprocessor*, which inputs a bunch of clauses and tries to simplify them. It uses all sorts of gimmicks that I didn't want to bother to include in the other programs. Finally, after reducing the problem until these gimmicks yield no further progress, it outputs an equivalent set of clauses that can be fed to a real solver.

If you have already read SAT0 (or some other program of this series), you might as well skip now past all the code for the "I/O wrapper," because you've seen it before—*except* for the new material in §2 below, which talks about a special file that makes it possible to undo the effects of preprocessing when constructing a solution to the original program.

The input on *stdin* is a series of lines with one clause per line. Each clause is a sequence of literals separated by spaces. Each literal is a sequence of one to eight ASCII characters between `!` and `}`, inclusive, not beginning with `~`, optionally preceded by `~` (which makes the literal "negative"). For example, Rivest's famous clauses on four variables, found in 6.5–(13) and 7.1.1–(32) of *TAOCP*, can be represented by the following eight lines of input:

```
x2 x3 ~x4
x1 x3 x4
~x1 x2 x4
~x1 ~x2 x3
~x2 ~x3 x4
~x1 ~x3 ~x4
x1 ~x2 ~x4
x1 x2 ~x3
```

Input lines that begin with `~␣` are ignored (treated as comments).

The running time in "mems" is also reported, together with the approximate number of bytes needed for data storage. One "mem" essentially means a memory access to a 64-bit word. (These totals don't include the time or space needed to parse the input or to format the output.)

**2.** One of the most important jobs of a preprocessor is to reduce the number of variables, if possible. But when that happens, and if the resulting clauses are satisfiable, the user often wants to know how to satisfy the original clauses; should the eliminated variables be true or false?

To answer such questions, this program produces an `erp` file, which reverses the effect of preprocessing. The `erp` file consists of zero or more groups of lines, one group for each eliminated variable. The first line of every group consists of the name of a literal (that is, the name of a variable, optionally preceded by `~`), followed by the three characters `␣<-`, followed by a number and end-of-line. That literal represents an eliminated variable or its negation.

The number after `<-`, say $k$, tells how many other lines belong to the same group. Those $k$ lines each contain a clause in the normal way, where the clauses can involve any variables that haven't been eliminated. The meaning is, "If all $k$ of these clauses are satisfied, by the currently known assignment to uneliminated variables, the literal should be true; otherwise it should be false."

A companion program, SAT12-ERP, reads an `erp` file together with the literals output by a SAT-solver, and assigns values to the eliminated variables by essentially processing the `erp` file *backwards*.

For example, SAT12-ERP would process the following simple three-line file

```
~x <-1
~y z
y <-0
```

by first setting `y` true, and then setting `x` to the complement of the value of `z`.

(Fine point: A SAT solver might not have actually given a value to `z` in this example, if the solved clauses could be satisfied regardless of whether `z` is true or false. In such cases SAT12-ERP would arbitrarily make `z` true and `x` false.)

Sometimes, as in the case of Rivest's axioms above, SAT12 will reduce the given clauses to the null set by eliminating all variables. Then SAT12-ERP will be able to exhibit a solution by examining the `erp` file alone, and no solver will be needed.

The `erp` file will be `/tmp/erp` unless another name is specified.

**3.** So here's the structure of the program. (Skip ahead if you are impatient to see the interesting stuff.)

```
#define o mems++    /* count one mem */
#define oo mems += 2    /* count two mems */
#define ooo mems += 3    /* count three mems */
#define O "%"    /* used for percent signs in format strings */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_flip.h"
  typedef unsigned int uint;    /* a convenient abbreviation */
  typedef unsigned long long ullng;    /* ditto */
  ⟨Type definitions 6⟩;
  ⟨Global variables 4⟩;
  ⟨Subroutines 26⟩;
  main(int argc, char *argv[])
  {
    register uint aa, b, c, cc, h, i, j, k, l, ll, p, pp, q, qq, r, s, t, u, uu, v, vv, w, ww, x;
    register uint rbits = 0;    /* random bits generated but not yet used */
    register ullng bits;
    register specialcase;
    ⟨Process the command line 5⟩;
    ⟨Initialize everything 9⟩;
    ⟨Input the clauses 10⟩;
    if (verbose & show_basics) ⟨Report the successful completion of the input phase 22⟩;
    ⟨Set up the main data structures 40⟩;
    imems = mems, mems = 0;
    ⟨Preprocess until everything is stable 91⟩;
  finish_up: ⟨Output the simplified clauses 97⟩;
    if (verbose & show_basics) {
      fprintf(stderr, "Altogether␣"O"llu+"O"llu␣mems,␣"O"llu␣bytes,␣"O"u␣cells;\n",
          imems, mems, bytes, xcells);
      if (sub_total + str_total)
        fprintf(stderr, "␣"O"u␣subsumption"O"s,␣"O"u␣strengthening"O"s.\n", sub_total,
            sub_total ≠ 1 ? "s" : "", str_total, str_total ≠ 1 ? "s" : "");
      fprintf(stderr, "␣false␣hit␣rates␣"O".3f␣of␣"O"llu,␣"O".3f␣of␣"O"llu.\n",
          sub_tries ? (double) sub_false/(double) sub_tries : 0.0, sub_tries,
          str_tries ? (double) str_false/(double) str_tries : 0.0, str_tries);
      if (elim_tries) fprintf(stderr, "␣"O".3f␣functional␣dependencies␣among␣"O"llu␣trials.\n",
            (double) func_total/(double) elim_tries, elim_tries);
      fprintf(stderr, "erp␣data␣written␣to␣file␣"O"s.\n", erp_file_name);
    }
  }
```

```
4. #define show_basics 1      /* verbose code for basic stats */
#define show_rounds 2          /* verbose code to show each round of elimination */
#define show_details 4         /* verbose code for further commentary */
#define show_resolutions 8     /* verbose code for resolution logging */
#define show_lit_ids 16        /* verbose extra help for debugging */
#define show_subtrials 32      /* verbose code to show subsumption tests */
#define show_restrials 64      /* verbose code to show resolution tests */
#define show_initial_clauses 128  /* verbose code to show the input clauses */

⟨ Global variables 4 ⟩ ≡
  int random_seed = 0;      /* seed for the random words of gb_rand */
  int verbose = show_basics;     /* level of verbosity */
  int hbits = 8;     /* logarithm of the number of the hash lists */
  int buf_size = 1024;     /* must exceed the length of the longest input line */
  FILE *erp_file;     /* file to allow reverse preprocessing */
  char erp_file_name[100] = "/tmp/erp";     /* its name */
  ullng imems, mems;     /* mem counts */
  ullng bytes;     /* memory used by main data structures */
  uint xcells;     /* total number of mem cells used */
  int cutoff = 10;     /* heuristic cutoff for variable elimination */
  ullng optimism = 25;     /* don't try to eliminate if more than this must peter out */
  int buckets = 32;     /* buckets for variable elimination sorting */
  ullng mem_max = 100000;     /* lower bound on number of cells allowed in mem */
  uint sub_total, str_total;     /* count of subsumptions, strengthenings */
  ullng sub_tries, sub_false, str_tries, str_false;     /* stats on those algorithms */
  int maxrounds = #7fffffff;     /* give up after this many elimination rounds */
  ullng timeout = #1fffffffffffffff;     /* give up after this many mems */
  ullng elim_tries, func_total;     /* stats for elimination */
```

See also sections 8, 39, and 81.

This code is used in section 3.

**5.** On the command line one can specify nondefault values for any of the following parameters:

- '`v`⟨ integer ⟩' to enable various levels of verbose output on *stderr*.
- '`h`⟨ positive integer ⟩' to adjust the hash table size.
- '`b`⟨ positive integer ⟩' to adjust the size of the input buffer.
- '`s`⟨ integer ⟩' to define the seed for any random numbers that are used.
- '`e`⟨ filename ⟩' to change the name of the `erp` output file.
- '`m`⟨ integer ⟩' to specify a minimum *mem* size (cell memory).
- '`c`⟨ integer ⟩' to specify a heuristic cutoff for degrees of variables to eliminate.
- '`C`⟨ integer ⟩' to specify a heuristic cutoff for excess of $pq$ versus $p + q$ when eliminating a variable that requires $pq$ resolutions.
- '`B`⟨ integer ⟩' to specify the maximum degree that is distinguished when ranking variables by degree.
- '`t`⟨ integer ⟩' to specify the maximum number of rounds of variable elimination that will be attempted. (In particular, '`t0`' will not eliminate any variables by resolution, although pure literals will go away.)
- '`T`⟨ integer ⟩' to set *timeout*: This program will stop preprocessing if it discovers that $mems > timeout$.

```
⟨ Process the command line 5 ⟩ ≡
  for (j = argc − 1, k = 0; j; j−−)
    switch (argv[j][0]) {
    case 'v': k |= (sscanf(argv[j] + 1, ""O"d", &verbose) − 1); break;
    case 'h': k |= (sscanf(argv[j] + 1, ""O"d", &hbits) − 1); break;
    case 'b': k |= (sscanf(argv[j] + 1, ""O"d", &buf_size) − 1); break;
    case 's': k |= (sscanf(argv[j] + 1, ""O"d", &random_seed) − 1); break;
    case 'e': sprintf(erp_file_name, ""O".99s", argv[j] + 1); break;
    case 'm': k |= (sscanf(argv[j] + 1, ""O"llu", &mem_max) − 1); break;
    case 'c': k |= (sscanf(argv[j] + 1, ""O"d", &cutoff) − 1); break;
    case 'C': k |= (sscanf(argv[j] + 1, ""O"llu", &optimism) − 1); break;
    case 'B': k |= (sscanf(argv[j] + 1, ""O"d", &buckets) − 1); break;
    case 't': k |= (sscanf(argv[j] + 1, ""O"d", &maxrounds) − 1); break;
    case 'T': k |= (sscanf(argv[j] + 1, ""O"lld", &timeout) − 1); break;
    default: k = 1;     /* unrecognized command-line option */
    }
  if (k ∨ hbits < 0 ∨ hbits > 30 ∨ buf_size ≤ 0) {
    fprintf(stderr, "Usage:␣"O"s␣[v<n>]␣[h<n>]␣[b<n>]␣[s<n>]␣[efoo.erp]␣[m<n>]", argv[0]);
    fprintf(stderr, "␣[c<n>]␣[C<n>]␣[B<n>]␣[t<n>]]␣[T<n>]␣<␣foo.sat\n");
    exit(−1);
  }
  if (¬(erp_file = fopen(erp_file_name, "w"))) {
    fprintf(stderr, "I␣couldn't␣open␣file␣"O"s␣for␣writing!\n", erp_file_name);
    exit(−16);
  }
```

This code is used in section 3.

**6. The I/O wrapper.** The following routines read the input and absorb it into temporary data areas from which all of the "real" data structures can readily be initialized. My intent is to incorporate these routines into all of the SAT-solvers in this series. Therefore I've tried to make the code short and simple, yet versatile enough so that almost no restrictions are placed on the sizes of problems that can be handled. These routines are supposed to work properly unless there are more than $2^{32}-1 = 4{,}294{,}967{,}295$ occurrences of literals in clauses, or more than $2^{31}-1 = 2{,}147{,}483{,}647$ variables or clauses.

In these temporary tables, each variable is represented by four things: its unique name; its serial number; the clause number (if any) in which it has most recently appeared; and a pointer to the previous variable (if any) with the same hash address. Several variables at a time are represented sequentially in small chunks of memory called "vchunks," which are allocated as needed (and freed later).

```
#define vars_per_vchunk 341   /* preferably (2^k − 1)/3 for some k */
⟨ Type definitions 6 ⟩ ≡
  typedef union {
    char ch8[8];
    uint u2[2];
    ullng lng;
  } octa;
  typedef struct tmp_var_struct {
    octa name;   /* the name (one to eight ASCII characters) */
    uint serial;   /* 0 for the first variable, 1 for the second, etc. */
    int stamp;   /* m if positively in clause m; −m if negatively there */
    struct tmp_var_struct *next;   /* pointer for hash list */
  } tmp_var;
  typedef struct vchunk_struct {
    struct vchunk_struct *prev;   /* previous chunk allocated (if any) */
    tmp_var var[vars_per_vchunk];
  } vchunk;
```

See also sections 7, 25, 27, 28, and 29.

This code is used in section 3.

**7.** Each clause in the temporary tables is represented by a sequence of one or more pointers to the **tmp_var** nodes of the literals involved. A negated literal is indicated by adding 1 to such a pointer. The first literal of a clause is indicated by adding 2. Several of these pointers are represented sequentially in chunks of memory, which are allocated as needed and freed later.

```
#define cells_per_chunk 511   /* preferably 2^k − 1 for some k */
⟨ Type definitions 6 ⟩ +≡
  typedef struct chunk_struct {
    struct chunk_struct *prev;   /* previous chunk allocated (if any) */
    tmp_var *cell[cells_per_chunk];
  } chunk;
```

**8.** ⟨ Global variables 4 ⟩ +≡

```
char *buf;   /* buffer for reading the lines (clauses) of stdin */
tmp_var **hash;   /* heads of the hash lists */
uint hash_bits[93][8];   /* random bits for universal hash function */
vchunk *cur_vchunk;   /* the vchunk currently being filled */
tmp_var *cur_tmp_var;   /* current place to create new tmp_var entries */
tmp_var *bad_tmp_var;   /* the cur_tmp_var when we need a new vchunk */
chunk *cur_chunk;   /* the chunk currently being filled */
tmp_var **cur_cell;   /* current place to create new elements of a clause */
tmp_var **bad_cell;   /* the cur_cell when we need a new chunk */
ullng vars;   /* how many distinct variables have we seen? */
ullng clauses;   /* how many clauses have we seen? */
ullng nullclauses;   /* how many of them were null? */
ullng cells;   /* how many occurrences of literals in clauses? */
```

**9.** ⟨ Initialize everything 9 ⟩ ≡

```
gb_init_rand(random_seed);
buf = (char *) malloc(buf_size * sizeof(char));
if (¬buf) {
  fprintf(stderr, "Couldn't␣allocate␣the␣input␣buffer␣(buf_size="O"d)!\n", buf_size);
  exit(−2);
}
hash = (tmp_var **) malloc(sizeof(tmp_var) ≪ hbits);
if (¬hash) {
  fprintf(stderr, "Couldn't␣allocate␣"O"d␣hash␣list␣heads␣(hbits="O"d)!\n", 1 ≪ hbits, hbits);
  exit(−3);
}
for (h = 0; h < 1 ≪ hbits; h++) hash[h] = Λ;
```

See also section 15.

This code is used in section 3.

**10.** The hash address of each variable name has $h$ bits, where $h$ is the value of the adjustable parameter *hbits*. Thus the average number of variables per hash list is $n/2^h$ when there are $n$ different variables. A warning is printed if this average number exceeds 10. (For example, if $h$ has its default value, 8, the program will suggest that you might want to increase $h$ if your input has 2560 different variables or more.)

All the hashing takes place at the very beginning, and the hash tables are actually recycled before any SAT-solving takes place; therefore the setting of this parameter is by no means crucial. But I didn't want to bother with fancy coding that would determine $h$ automatically.

```
⟨Input the clauses 10⟩ ≡
  while (1) {
    if (¬fgets(buf, buf_size, stdin)) break;
    clauses++;
    if (buf[strlen(buf) − 1] ≠ '\n') {
      fprintf(stderr, "The␣clause␣on␣line␣"O"lld␣("O".20s...)␣is␣too␣long␣for␣me;\n", clauses,
          buf);
      fprintf(stderr, "␣my␣buf_size␣is␣only␣"O"d!\n", buf_size);
      fprintf(stderr, "Please␣use␣the␣command-line␣option␣b<newsize>.\n");
      exit(−4);
    }
    ⟨Input the clause in buf 11⟩;
  }
  if ((vars ≫ hbits) ≥ 10) {
    fprintf(stderr, "There␣are␣"O"lld␣variables␣but␣only␣"O"d␣hash␣tables;\n", vars, 1 ≪ hbits);
    while ((vars ≫ hbits) ≥ 10) hbits++;
    fprintf(stderr, "␣maybe␣you␣should␣use␣command-line␣option␣h"O"d?\n", hbits);
  }
  clauses −= nullclauses;
  if (clauses ≡ 0) {
    fprintf(stderr, "No␣clauses␣were␣input!\n");
    exit(−77);
  }
  if (vars ≥ #80000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣"O"llu␣variables!\n", vars);
    exit(−664);
  }
  if (clauses ≥ #80000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣"O"llu␣clauses!\n", clauses);
    exit(−665);
  }
  if (cells ≥ #100000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣"O"llu␣occurrences␣of␣literals!\n", cells);
    exit(−666);
  }
```

This code is used in section 3.

**11.** ⟨ Input the clause in *buf* 11 ⟩ ≡

```
for (j = k = 0; ; ) {
  while (buf[j] ≡ '␣') j++;     /* scan to nonblank */
  if (buf[j] ≡ '\n') break;
  if (buf[j] < '␣' ∨ buf[j] > '~') {
    fprintf(stderr, "Illegal␣character␣(code␣#"O"x)␣in␣the␣clause␣on␣line␣"O"lld!\n",
        buf[j], clauses);
    exit(−5);
  }
  if (buf[j] ≡ '~') i = 1, j++;
  else i = 0;
  ⟨ Scan and record a variable; negate it if i ≡ 1 12 ⟩;
}
if (k ≡ 0) {
  fprintf(stderr, "(Empty␣line␣"O"lld␣is␣being␣ignored)\n", clauses);
  nullclauses++;     /* strictly speaking it would be unsatisfiable */
}
goto clause_done;
empty_clause: ⟨ Remove all variables of the current clause 19 ⟩;
clause_done: cells += k;
```

This code is used in section 10.

**12.** We need a hack to insert the bit codes 1 and/or 2 into a pointer value.

```
#define hack_in(q, t) (tmp_var *)(t | (ullng) q)
```

⟨ Scan and record a variable; negate it if $i \equiv 1$ 12 ⟩ ≡

```
{
  register tmp_var *p;
  if (cur_tmp_var ≡ bad_tmp_var) ⟨ Install a new vchunk 13 ⟩;
  ⟨ Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 16 ⟩;
  ⟨ Find cur_tmp_var→name in the hash table at p 17 ⟩;
  if (p→stamp ≡ clauses ∨ p→stamp ≡ −clauses) ⟨ Handle a duplicate literal 18 ⟩
  else {
    p→stamp = (i ? −clauses : clauses);
    if (cur_cell ≡ bad_cell) ⟨ Install a new chunk 14 ⟩;
    *cur_cell = p;
    if (i ≡ 1) *cur_cell = hack_in(*cur_cell, 1);
    if (k ≡ 0) *cur_cell = hack_in(*cur_cell, 2);
    cur_cell++, k++;
  }
}
```

This code is used in section 11.

**13.** ⟨Install a new **vchunk** 13⟩ ≡

```
{
  register vchunk *new_vchunk;
  new_vchunk = (vchunk *) malloc(sizeof(vchunk));
  if (¬new_vchunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣vchunk!\n");
    exit(−6);
  }
  new_vchunk→prev = cur_vchunk, cur_vchunk = new_vchunk;
  cur_tmp_var = &new_vchunk→var[0];
  bad_tmp_var = &new_vchunk→var[vars_per_vchunk];
}
```

This code is used in section 12.

**14.** ⟨Install a new **chunk** 14⟩ ≡

```
{
  register chunk *new_chunk;
  new_chunk = (chunk *) malloc(sizeof(chunk));
  if (¬new_chunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣chunk!\n");
    exit(−7);
  }
  new_chunk→prev = cur_chunk, cur_chunk = new_chunk;
  cur_cell = &new_chunk→cell[0];
  bad_cell = &new_chunk→cell[cells_per_chunk];
}
```

This code is used in section 12.

**15.** The hash code is computed via "universal hashing," using the following precomputed tables of random bits.

⟨Initialize everything 9⟩ +≡

```
for (j = 92; j; j−−)
  for (k = 0; k < 8; k++) hash_bits[j][k] = gb_next_rand();
```

**16.** ⟨Put the variable name beginning at *buf*[$j$] in *cur_tmp_var→name* and compute its hash code $h$ 16⟩ ≡

```
cur_tmp_var→name.lng = 0;
for (h = l = 0; buf[j + l] > '␣' ∧ buf[j + l] ≤ '~'; l++) {
  if (l > 7) {
    fprintf(stderr, "Variable␣name␣"O".9s...␣in␣the␣clause␣on␣line␣"O"lld␣is␣too␣long!\n",
        buf + j, clauses);
    exit(−8);
  }
  h ⊕= hash_bits[buf[j + l] − '!'][l];
  cur_tmp_var→name.ch8[l] = buf[j + l];
}
if (l ≡ 0) goto empty_clause;    /* '~' by itself is like 'true' */
j += l;
h &= (1 ≪ hbits) − 1;
```

This code is used in section 12.

**17.** ⟨ Find *cur_tmp_var→name* in the hash table at *p* 17 ⟩ ≡

```
for (p = hash[h]; p; p = p→next)
  if (p→name.lng ≡ cur_tmp_var→name.lng) break;
if (¬p) {    /* new variable found */
  p = cur_tmp_var++;
  p→next = hash[h], hash[h] = p;
  p→serial = vars++;
  p→stamp = 0;
}
```

This code is used in section 12.

**18.** The most interesting aspect of the input phase is probably the "unwinding" that we might need to do when encountering a literal more than once in the same clause.

⟨ Handle a duplicate literal 18 ⟩ ≡

```
{
  if ((p→stamp > 0) ≡ (i > 0)) goto empty_clause;
}
```

This code is used in section 12.

**19.** An input line that begins with '~␣' is silently treated as a comment. Otherwise redundant clauses are logged, in case they were unintentional. (One can, however, intentionally use redundant clauses to force the order of the variables.)

⟨ Remove all variables of the current clause 19 ⟩ ≡

```
while (k) {
  ⟨ Move cur_cell backward to the previous cell 20 ⟩;
  k−−;
}
if ((buf[0] ≠ '~') ∨ (buf[1] ≠ '␣'))
  fprintf(stderr, "(The␣clause␣on␣line␣"O"lld␣is␣always␣satisfied)\n", clauses);
nullclauses++;
```

This code is used in section 11.

**20.** ⟨ Move *cur_cell* backward to the previous cell 20 ⟩ ≡

```
if (cur_cell > &cur_chunk→cell[0]) cur_cell−−;
else {
  register chunk *old_chunk = cur_chunk;

  cur_chunk = old_chunk→prev; free(old_chunk);
  bad_cell = &cur_chunk→cell[cells_per_chunk];
  cur_cell = bad_cell − 1;
}
```

This code is used in sections 19 and 43.

**21.** ⟨ Move *cur_tmp_var* backward to the previous temporary variable 21 ⟩ ≡

```
if (cur_tmp_var > &cur_vchunk→var[0]) cur_tmp_var−−;
else {
  register vchunk *old_vchunk = cur_vchunk;

  cur_vchunk = old_vchunk→prev; free(old_vchunk);
  bad_tmp_var = &cur_vchunk→var[vars_per_vchunk];
  cur_tmp_var = bad_tmp_var − 1;
}
```

This code is used in section 44.

**22.** ⟨ Report the successful completion of the input phase 22 ⟩ ≡

*fprintf* (*stderr*, "("*O*"lld␣variables,␣"*O*"lld␣clauses,␣"*O*"llu␣literals␣successfully␣read)\n",
*vars*, *clauses*, *cells*);

This code is used in section 3.

**23. SAT preprocessing.** This program applies transformations that either reduce the number of clauses or keep that number fixed while reducing the number of variables. In this process we might wind up with no clauses whatsoever (thus showing that the problem is satisfiable), or we might wind up deducing an empty clause (thus showing that the problem is unsatisfiable). But since our transformations always go "downhill," we can't solve really tough problems in this way. Our main goal is to make other SAT-solvers more efficient, by using transformation-oriented data structures that would not be appropriate for them.

Of course we remove all unit clauses, by forcing the associated literal to be true. Every clause that's eventually output by this program will have length two or more.

More generally, we remove all clauses that are subsumed by other clauses: If every literal in clause $C$ appears also in another clause $C'$, we remove $C'$. In particular, duplicate clauses are discarded.

We also remove "pure literals," which occur with only one sign. More generally, if variable $x$ occurs positively $a$ times and negatively $b$ times, we eliminate $x$ by resolution whenever $ab \le a + b$, because resolution will replace those $a+b$ clauses by at most $ab$ clauses that contain neither $x$ nor $\bar{x}$. That happens whenever $(a-1)(b-1) \le 1$, thus not only when $a = 0$ or $b = 0$ but also when $a = 1$ or $b = 1$ or $a = b = 2$.

Furthermore, we try resolution even when $ab > a + b$, because resolution often produces fewer than $ab$ new clauses (especially when subsumed clauses are removed). We don't try it, however, when $a$ and $b$ both exceed a user-specified cutoff parameter.

Another nice case, "strengthening" or "self-subsumption," arises when clause $C$ *almost* subsumes another clause $C'$, except that $\bar{x}$ occurs in $C$ while $x$ occurs in $C'$; every *other* literal of $C$ does appear in $C'$. In such cases we can remove $x$ from $C'$, because $C' \setminus x = C \diamond C'$.

**24.** I haven't spent much time trying to design data structures that are optimum for the operations needed by this program; some form of ZDD might well be better for subsumption, depending on the characteristics of the clauses that are given. But I think the fairly simple structures used here will be adequate.

First, this program keeps all of the clause information in a quadruply linked structure like that of dancing links: Each cell is in a doubly linked vertical list of all cells for a particular literal, as well as in a doubly linked horizontal list of all cells for a particular clause.

Second, each clause has a 64-bit "signature" containing 1s for hash codes of its literals. This signature speeds up subsumption testing.

In some cases there's a sequential scan through all variables or through all clauses. With fancier data structures I could add extra techniques to skip more quickly over variables and clauses that have been eliminated or dormant; but those structures have their own associated costs. As usual, I've tried to balance simplicity and efficiency, using my best guess about how important each operation will be in typical cases. (For example, I don't mind making several passes over the data, if each previous pass has brought rich rewards.)

Two main lists govern the operations of highest priority: The "to-do stack" contains variables whose values can readily be fixed or ignored; the "strengthened stack" contains clauses that have become shorter. The program tries to keep the to-do stack empty at most times, because that operation is cheap and productive. And when the to-do stack is empty, it's often a good idea to clear off the strengthened stack by seeing if any of its clauses subsume or strengthen others.

As in other programs of this series, I eschew pointer variables, which are implemented inefficiently by the programming environment of my 64-bit machine. Instead, links between items of data are indices into arrays of structured records. The only downside of this policy is that I need to decide in advance how large those arrays should be.

**25.** The main *mem* array contains **cell** structs, each occupying three octabytes. Every literal of every clause appears in a cell, with six 32-bit fields to identify the literal and clause together with local *left*/*right* links for that clause and local *up*/*down* links for that literal.

The first two cells, *mem*[0] are *mem*[1], are reserved for special purposes.

The next cells, *mem*[2] through *mem*[$2n+1$] if there are $n$ variables initially, are heads of the literal lists, identifiable by their location. Such cells have a 64-bit signature field instead of *left*/*right* links; this field contains the literal's hash code.

The next cells, *mem*[$2n+2$] through *mem*[$2n+m+1$] if there are $m$ clauses initially, are heads of the clause lists, identifiable by their location. Such cells have a 64-bit signature field instead of *up*/*down* links; this field is the bitwise OR of the hash codes of the clauses's literals.

All remaining cells, from *mem*[$2n+m+2$] through *mem*[*mem_max* $-1$], either contain elements of clauses or are currently unused.

Because of the overlap between 32-bit and 64-bit fields, a *cell* struct is defined in terms of the union type **octa**. Macros are defined to facilitate references to the individual fields in different contexts.

```
#define is_lit(k) ((k) < lit_head_top)
#define is_cls(k) ((k) < cls_head_top)
#define up(k) mem[k].litinf.u2[0]  /* next "higher" clause of same literal */
#define down(k) mem[k].litinf.u2[1]  /* next "lower" clause of same literal */
#define left(k) mem[k].clsinf.u2[0]  /* next smaller literal of same clause */
#define right(k) mem[k].clsinf.u2[1]  /* next larger literal of same clause */
#define litsig(k) mem[k].clsinf.lng  /* hash signature of a literal */
#define clssig(k) mem[k].litinf.lng  /* hash signature of a clause */
#define occurs(l) mem[l].lit  /* how many clauses contain l? */
#define littime(l) mem[l].cls  /* what's their most recent creation time? */
#define size(c) mem[c].cls  /* how many literals belong to c? */
#define clstime(c) mem[c].lit  /* most recent full exploitation of c */
```

⟨Type definitions 6⟩ +≡

```
typedef struct cell_struct {
  uint lit;  /* literal number (except in list heads) */
  uint cls;  /* clause number (except in list heads) */
  octa litinf, clsinf;  /* links within literal and clause lists */
} cel;  /* I'd call this cell except for confusion with cell fields */
```

**26.** Here's a way to display a cell symbolically when debugging with GDB (which doesn't see those macros):

⟨Subroutines 26⟩ ≡

```
void show_cell(uint k)
{
  fprintf(stderr, "mem["O"u]=", k);
  if (is_lit(k)) fprintf(stderr, "occ␣"O"u,␣time␣"O"u,␣sig␣"O"llx,␣up␣"O"u,␣dn␣"O"u\n",
        occurs(k), littime(k), litsig(k), up(k), down(k));
  else if (is_cls(k))
    fprintf(stderr, "size␣"O"u,␣time␣"O"u,␣sig␣"O"llx,␣left␣"O"u,␣right␣"O"u\n", size(k),
        clstime(k), clssig(k), left(k), right(k));
  else fprintf(stderr, "lit␣"O"u,␣cls␣"O"u,␣lft␣"O"u,␣rt␣"O"u,␣up␣"O"u,␣dn␣"O"u\n",
        mem[k].lit, mem[k].cls, left(k), right(k), up(k), down(k));
}
```

See also sections 30, 31, 32, 33, 37, 38, and 98.

This code is used in section 3.

**27.** The *vmem* array contains global information about individual variables. Variable number $k$, for $1 \le k \le n$, corresponds to the literals numbered $2k$ and $2k+1$.

Variables that are on the "to-do stack" of easy pickings (newly discovered unit clauses and pure literals) have a nonzero *status* field. The to-do stack begins at *to_do* and ends at 0. The *status* field is *forced_true* or *forced_false* if the variable is to be set true or false, respectively; or it is *elim_quiet* if the variable is simply supposed to be eliminated quietly.

Sometimes a variable is eliminated via resolution, without going onto the to-do stack. In such cases its *status* is *elim_res*.

Each variable also has an *stable* field, which is nonzero if the variable has not been involved in recent transformations.

We add a 16-bit *spare* field, and a 32-bit filler field, so that a **variable** struct fills three octabytes.

**#define** *thevar*$(l)$ $((l) \gg 1)$
**#define** *litname*$(l)$ $(l)$ & 1 ? "~" : "", *vmem*[*thevar*$(l)$].*name*.*ch8* /* used in printouts */
**#define** *pos_lit*$(v)$ $((v) \ll 1)$
**#define** *neg_lit*$(v)$ $(((v) \ll 1) + 1)$
**#define** *bar*$(l)$ $((l) \oplus 1)$ /* the complement of $l$ */
**#define** *touch*$(w)$ $o$, *vmem*[*thevar*$(w)$].*stable* $= 0$
**#define** *norm* 0
**#define** *elim_quiet* 1
**#define** *elim_res* 2
**#define** *forced_true* 3
**#define** *forced_false* 4

⟨Type definitions 6⟩ +≡
```
typedef struct var_struct {
  octa name;    /* the variable's symbolic name */
  uint link;    /* pointer for the to-do stack */
  char status;  /* current status */
  char stable;  /* not recently touched? */
  short spare;  /* filler */
  uint blink;   /* link for a bucket list list */
  uint filler;  /* another filler */
} variable;
```

**28.** Three octabytes doesn't seem quite enough for the data associated with each literal. So here's another struct to handle the extra stuff.

⟨Type definitions 6⟩ +≡
```
typedef struct lit_struct {
  ullng extra;  /* useful in the elimination routine */
} literal;
```

**29.** Similarly, each clause needs more elbow room.

The stack of strengthened clauses begins at *strengthened* and ends at *sentinel*. Clause $c$ is on this list if and only if *slink*$(c)$ is nonzero.

**#define** *sentinel* 1
**#define** *slink*$(c)$ *cmem*[$c -$ *lit_head_top*].*link*
**#define** *newsize*$(c)$ *cmem*[$c -$ *lit_head_top*].*size*

⟨Type definitions 6⟩ +≡
```
typedef struct cls_struct {
  uint link;  /* next clause in the strengthened list, or zero */
  uint size;  /* data for clause subsumption/strengthening */
} clause;
```

**30.** Here's a subroutine that prints clause number $c$.

Note that the number of a clause is its position in *mem*, which is somewhat erratic. Initially that position is $2n+1$ greater than the clause's position in the input; for example, if there are 100 variables, the first clause that was input will be internal clause number 202. As computation proceeds, however, we might decide to change a clause's number at any time.

⟨ Subroutines 26 ⟩ +≡

```
void print_clause(int c)
{
  register uint k, l;
  if (is_cls(c) ∧ ¬is_lit(c)) {
    if (¬size(c)) return;
    fprintf(stderr, ""O"d:", c);     /* show the clause number */
    for (k = right(c); ¬is_cls(k); k = right(k)) {
      l = mem[k].lit;
      fprintf(stderr, "␣"O"s"O".8s", litname(l));
      if (verbose & show_lit_ids) fprintf(stderr, "("O"u)", l);
    }
    fprintf(stderr, "\n");
  } else fprintf(stderr, "there␣is␣no␣clause␣"O"d!\n", c);
}
```

**31.** Another subroutine shows all the clauses that are currently in memory.

⟨ Subroutines 26 ⟩ +≡

```
void print_all(void)
{
  register uint c;
  for (c = lit_head_top; is_cls(c); c++)
    if (size(c)) print_clause(c);
}
```

**32.** With a similar subroutine we can print out all of the clauses that involve a particular literal.

⟨ Subroutines 26 ⟩ +≡

```
void print_clauses_for(int l)
{
  register uint k;
  if (is_lit(l) ∧ l ≥ 2) {
    if (vmem[thevar(l)].status) {
      fprintf(stderr, "␣"O"s␣has␣been␣%s!\n", vmem[thevar(l)].name.ch8,
          vmem[thevar(l)].status ≡ elim_res ? "eliminated" : vmem[thevar(l)].status ≡ elim_quiet ?
          "quietly␣eliminated" : vmem[thevar(l)].status ≡ forced_true ? "forced␣true" :
          vmem[thevar(l)].status ≡ forced_false ? "forced␣false" : "clobbered");
      return;
    }
    fprintf(stderr, "␣"O"s"O".8s", litname(l));
    if (verbose & show_lit_ids) fprintf(stderr, "("O"u)", l);
    fprintf(stderr, "␣is␣in");
    for (k = down(l); ¬is_lit(k); k = down(k)) fprintf(stderr, "␣"O"u", mem[k].cls);
    fprintf(stderr, "\n");
  } else fprintf(stderr, "There␣is␣no␣literal␣"O"d!\n", l);
}
```

**33.** Speaking of debugging, here's a routine to check if the links in *mem* have gone awry.

```
#define sanity_checking 0   /* set this to 1 if you suspect a bug */
⟨ Subroutines 26 ⟩ +≡
  void sanity(void)
  {
    register uint l, k, c, countl, countc, counta, s;
    register ullng bits;
    for (l = 2, countl = 0; is_lit(l); l++)
      if (vmem[thevar(l)].status ≡ norm) ⟨ Verify the cells for literal l 34 ⟩;
    for (c = l, countc = 0; is_cls(c); c++)
      if (size(c)) ⟨ Verify the cells for clause c 35 ⟩;
    if (countl ≠ countc ∧ to_do ≡ 0)
      fprintf(stderr, ""O"u␣cells␣in␣lit␣lists␣but␣"O"u␣cells␣in␣cls␣lists!\n", countl, countc);
    ⟨ Check the avail list 36 ⟩;
    if (xcells ≠ cls_head_top + countc + counta + 1)
      fprintf(stderr, "memory␣leak␣of␣"O"d␣cells!\n", (int)(xcells − cls_head_top − countc − counta − 1));
  }
```

**34.** ⟨ Verify the cells for literal *l* 34 ⟩ ≡

```
  {
    for (k = down(l), s = 0; ¬is_lit(k); k = down(k)) {
      if (k ≥ xcells) {
        fprintf(stderr, "address␣in␣lit␣list␣"O"u␣out␣of␣range!\n", l);
        goto bad_l;
      }
      if (mem[k].lit ≠ l)
        fprintf(stderr, "literal␣wrong␣at␣cell␣"O"u␣("O"u␣not␣"O"u)!\n", k, mem[k].lit, l);
      if (down(up(k)) ≠ k) {
        fprintf(stderr, "down/up␣link␣wrong␣at␣cell␣"O"u␣of␣lit␣list␣"O"u!\n", k, l);
        goto bad_l;
      }
      countl++, s++;
    }
    if (k ≠ l) fprintf(stderr, "lit␣list␣"O"u␣ends␣at␣"O"u!\n", l, k);
    else if (down(up(k)) ≠ k) fprintf(stderr, "down/up␣link␣wrong␣at␣lit␣list␣head␣"O"u!\n", l);
    if (s ≠ occurs(l))
      fprintf(stderr, "literal␣"O"u␣occurs␣in␣"O"u␣clauses,␣not␣"O"u!\n", l, s, occurs(l));
  bad_l: continue;
  }
```

This code is used in section 33.

**35.** The literals of a clause must appear in increasing order.

```
⟨ Verify the cells for clause c 35 ⟩ ≡
  {
    bits = 0;
    for (k = right(c), l = s = 0; ¬is_cls(k); k = right(k)) {
      if (k ≥ xcells) {
        fprintf(stderr, "address␣in␣cls␣list␣"O"u␣out␣of␣range!\n", c);
        goto bad_c;
      }
      if (mem[k].cls ≠ c)
        fprintf(stderr, "clause␣wrong␣at␣cell␣"O"u␣("O"u␣not␣"O"u)!\n", k, mem[k].cls, c);
      if (right(left(k)) ≠ k) {
        fprintf(stderr, "right/left␣link␣wrong␣at␣cell␣"O"u␣of␣cls␣list␣"O"u!\n", k, c);
        goto bad_c;
      }
      if (thevar(mem[k].lit) ≤ thevar(l))
        fprintf(stderr, "literals␣"O"u␣and␣"O"u␣out␣of␣order␣in␣cell␣"O"u␣of␣clause␣"O"u!\n",
            l, mem[k].lit, k, c);
      l = mem[k].lit;
      bits |= litsig(l);
      countc++, s++;
    }
    if (k ≠ c) fprintf(stderr, "cls␣list␣"O"u␣ends␣at␣"O"u!\n", c, k);
    else if (right(left(k)) ≠ k)
      fprintf(stderr, "right/left␣link␣wrong␣of␣cls␣list␣head␣"O"u!\n", c);
    if (bits ≠ clssig(c)) fprintf(stderr, "signature␣wrong␣at␣clause␣"O"u!\n", c);
    if (s ≠ size(c)) fprintf(stderr, "clause␣"O"u␣has␣"O"u␣literals,␣not␣"O"u!\n", c, s, size(c));
  bad_c: continue;
  }
```

This code is used in section 33.

**36.** Unused cells of *mem* either lie above *xcells* or appear in the *avail* stack. Entries of the latter list are linked together by *left* links, terminated by 0; their other fields are undefined.

```
⟨ Check the avail list 36 ⟩ ≡
  for (k = avail, counta = 0; k; k = left(k)) {
    if (k ≥ xcells ∨ is_cls(k)) {
      fprintf(stderr, "address␣out␣of␣range␣in␣avail␣stack!\n");
      break;
    }
    counta++;
  }
```

This code is used in section 33.

**37.** Of course we need the usual memory allocation routine, to deliver a fresh cell when needed.

(The author fondly recalls the day in autumn, 1960, when he first learned about linked lists and the associated *avail* stack, while reading the program for the BALGOL compiler on the Burroughs 220 computer.)

```
⟨ Subroutines 26 ⟩ +≡
  uint get_cell(void)
  {
    register uint k;
    if (avail) {
      k = avail;
      o, avail = left(k);
      return k;
    }
    if (xcells ≡ mem_max) {
      fprintf(stderr, "Oops,␣we're␣out␣of␣memory␣(mem_max="O"llu)!\nTry␣option␣m.\n",
          mem_max);
      exit(−9);
    }
    return xcells++;
  }
```

**38.** Conversely, we need quick ways to recycle cells that have done their duty.

```
⟨ Subroutines 26 ⟩ +≡
  void free_cell(uint k)
  {
    o, left(k) = avail;   /* the free_cell routine shouldn't change anything else in mem[k] */
    avail = k;
  }
  void free_cells(uint k, uint kk)
  {   /* k = kk or left(kk) or left(left(kk)), etc. */
    o, left(k) = avail;
    avail = kk;
  }
```

**39.** ⟨ Global variables 4 ⟩ +≡

```
  cel *mem;   /* the master array of cells */
  uint lit_head_top;   /* first cell not in a literal list head */
  uint cls_head_top;   /* first cell not in a clause list head */
  uint avail;   /* top of the stack of available cells */
  uint to_do;   /* top of the to-do stack */
  uint strengthened;   /* top of the strengthened stack */
  variable *vmem;   /* auxiliary data for variables */
  literal *lmem;   /* auxiliary data for literals */
  clause *cmem;   /* auxiliary data for clauses */
  int vars_gone;   /* we've eliminated this many variables so far */
  int clauses_gone;   /* we've eliminated this many clauses so far */
  uint time;   /* the number of rounds of variable elimination we've done */
```

**40. Initializing the real data structures.** We're ready now to convert the temporary chunks of data into the form we want, and to recycle those chunks. The code below is, of course, hacked from what has worked in previous programs of this series.

```
⟨ Set up the main data structures 40 ⟩ ≡
  ⟨ Allocate the main arrays 41 ⟩;
  ⟨ Copy all the temporary cells to the mem array in proper format 42 ⟩;
  ⟨ Copy all the temporary variable nodes to the vmem array in proper format 44 ⟩;
  ⟨ Check consistency 45 ⟩;
  ⟨ Finish building the cell data structures 46 ⟩;
  ⟨ Allocate the subsidiary arrays 52 ⟩;
```

This code is used in section 3.

**41.** There seems to be no good way to predict how many cells we'll need, because the size of clauses can grow exponentially as the number of clauses shrinks. Here we allow for twice the number of cells in the input, or the user-supplied value of *mem_max*, whichever is larger—provided that we don't exceed 32-bit addresses.

```
⟨ Allocate the main arrays 41 ⟩ ≡
  free(buf); free(hash);   /* a tiny gesture to make a little room */
  lit_head_top = vars + vars + 2;
  cls_head_top = lit_head_top + clauses;
  xcells = cls_head_top + cells + 1;
  if (xcells + cells > mem_max) mem_max = xcells + cells;
  if (mem_max ≥ #100000000) mem_max = #ffffffff;
  mem = (cel *) malloc(mem_max * sizeof(cel));
  if (¬mem) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣big␣mem␣array!\n");
    exit(−10);
  }
  bytes = mem_max * sizeof(cel);
  vmem = (variable *) malloc((vars + 1) * sizeof(variable));
  if (¬vmem) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣vmem␣array!\n");
    exit(−11);
  }
  bytes += (vars + 1) * sizeof(variable);
```

This code is used in section 40.

**42.** ⟨ Copy all the temporary cells to the *mem* array in proper format 42 ⟩ ≡

```
  for (l = 2; is_lit(l); l++) o, down(l) = l;
  for (c = clauses, j = cls_head_top; c; c−−) {
    ⟨ Insert the cells for the literals of clause c 43 ⟩;
  }
  if (j ≠ cls_head_top + cells) {
    fprintf(stderr, "Oh␣oh,␣something␣happened␣to␣"O"d␣cells!\n", (int)(cls_head_top + cells − j));
    exit(−15);
  }
```

This code is used in section 40.

**43.** The basic idea is to "unwind" the steps that we went through while building up the chunks.

```
#define hack_out(q) (((ullng) q) & #3)
#define hack_clean(q) ((tmp_var *)((ullng) q & -4))
⟨ Insert the cells for the literals of clause c 43 ⟩ ≡
  for (i = 0; i < 2; ) {
    ⟨ Move cur_cell backward to the previous cell 20 ⟩;
    i = hack_out(*cur_cell);
    p = hack_clean(*cur_cell)→serial;
    p += p + (i & 1) + 2;
    o, mem[j].lit = p, mem[j].cls = cc = c + lit_head_top − 1;
    ooo, down(j) = down(p), down(p) = j++;
  }
  o, left(cc) = cc;
```

This code is used in section 42.

**44.**

```
⟨ Copy all the temporary variable nodes to the vmem array in proper format 44 ⟩ ≡
  for (c = vars; c; c−−) {
    ⟨ Move cur_tmp_var backward to the previous temporary variable 21 ⟩;
    o, vmem[c].name.lng = cur_tmp_var→name.lng;
    o, vmem[c].stable = vmem[c].status = 0;
  }
```

This code is used in section 40.

**45.** We should now have unwound all the temporary data chunks back to their beginnings.

```
⟨ Check consistency 45 ⟩ ≡
  if (cur_cell ≠ &cur_chunk→cell[0] ∨ cur_chunk→prev ≠ Λ ∨ cur_tmp_var ≠
        &cur_vchunk→var[0] ∨ cur_vchunk→prev ≠ Λ) confusion("consistency");
  free(cur_chunk); free(cur_vchunk);
```

This code is used in section 40.

**46.**

```
⟨ Finish building the cell data structures 46 ⟩ ≡
  for (l = 2; is_lit(l); l++) ⟨ Set the up links for l and the left links of its cells 47 ⟩;
  for (c = l; is_cls(c); c++) ⟨ Set the right links for c, and its signature and size 49 ⟩;
```

This code is used in section 40.

**47.** Since we process the literal lists in order, each clause is automatically sorted, with its literals appearing in increasing order from left to right. (That fact will help us significantly when we test for subsumption or compute resolvents.)

The clauses of a *literal*'s list are initially in order too. But we *don't* attempt to preserve that. Clauses will soon get jumbled.

⟨ Set the *up* links for $l$ and the *left* links of its cells 47 ⟩ ≡

```
{
  for (j = l, k = down(j), s = 0; ¬is_lit(k); o, j = k, k = down(j)) {
    o, up(k) = j;
    o, c = mem[k].cls;
    ooo, left(k) = left(c), left(c) = k;
    s++;
  }
  if (k ≠ l) confusion("lit␣init");
  o, occurs(l) = s, littime(l) = 0;
  o, up(l) = j;
  if (s ≡ 0) {
    w = l;
    if (verbose & show_details)
      fprintf(stderr, "no␣input␣clause␣contains␣the␣literal␣"O"s"O".8s\n", litname(w));
    ⟨ Set literal w to false unless it's already set 51 ⟩;
  } else ⟨ Set litsig(l) 48 ⟩;
}
```

This code is used in section 46.

**48.** I'm using two hash bits here, because experiments showed that this policy was almost always better than to use a single hash bit.

As in other programs of this series, I assume that it costs four mems to generate 31 new random bits.

⟨ Set *litsig*($l$) 48 ⟩ ≡

```
{
  if (rbits < #40) mems += 4, rbits = gb_next_rand() | (1U ≪ 30);
  o, litsig(l) = 1LLU ≪ (rbits & #3f);
  rbits ≫= 6;
  if (rbits < #40) mems += 4, rbits = gb_next_rand() | (1U ≪ 30);
  o, litsig(l) |= 1LLU ≪ (rbits & #3f);
  rbits ≫= 6;
}
```

This code is used in section 47.

**49.** ⟨ Set the *right* links for $c$, and its signature and size 49 ⟩ ≡

```
{
  bits = 0;
  for (j = c, k = left(j), s = 0; ¬is_cls(k); o, j = k, k = left(k)) {
    o, right(k) = j;
    o, w = mem[k].lit;
    o, bits |= litsig(w);
    s++;
  }
  if (k ≠ c) confusion("cls␣init");
  o, size(c) = s, clstime(c) = 0;
  oo, clssig(c) = bits, right(c) = j;
  if (s ≤ 1) {
    if (s ≡ 0) confusion("empty␣clause");
    if (verbose & show_details)
      fprintf(stderr, "clause␣"O"u␣is␣the␣single␣literal␣"O"s"O".8s\n", c, litname(w));
    ⟨ Force literal w to be true 50 ⟩;
  }
}
```

This code is used in section 46.

**50.** Here we assume that *thevar*($w$) hasn't already been eliminated. A unit clause has arisen, with $w$ as its only literal.

A variable might be touched after it has been put into the to-do stack. Thus we can't call it stable yet, even though its value won't change.

⟨ Force literal $w$ to be true 50 ⟩ ≡

```
{
  register int k = thevar(w);
  if (w & 1) {
    if (o, vmem[k].status ≡ norm) {
      o, vmem[k].status = forced_false;
      vmem[k].link = to_do, to_do = k;
    } else if (vmem[k].status ≡ forced_true) goto unsat;
  } else {
    if (o, vmem[k].status ≡ norm) {
      o, vmem[k].status = forced_true;
      vmem[k].link = to_do, to_do = k;
    } else if (vmem[k].status ≡ forced_false) goto unsat;
  }
}
```

This code is used in sections 49, 54, 64, and 90.

**51.** The logic in this step is similar to the previous one, except that we aren't *forcing* a value: Either $w$ wasn't present in any of the original clauses, or its final occurrence has disappeared.

It's possible that all occurrences of $\bar{w}$ have already disappeared too. In that case (which arises if and only if *thevar*($w$) is already on the to-do list at this point, and its *status* indicates that $w$ has been forced true), we just change the status to *elim_quiet*, because the variable needn't be set either true or false.

⟨Set literal $w$ to *false* unless it's already set 51⟩ ≡

```
{
  register int k = thevar(w);
  if (o, vmem[k].status ≡ norm) {
    o, vmem[k].status = (w & 1 ? forced_true : forced_false);
    vmem[k].link = to_do, to_do = k;
  } else if (vmem[k].status ≡ (w & 1 ? forced_false : forced_true))
    o, vmem[k].status = elim_quiet, vmem[k].stable = 1;
}
```

This code is used in sections 47, 56, 60, 63, and 88.

**52.** ⟨Allocate the subsidiary arrays 52⟩ ≡

```
lmem = (literal *) malloc(lit_head_top * sizeof(literal));
if (¬lmem) {
  fprintf(stderr, "Oops, I can't allocate the lmem array!\n");
  exit(−12);
}
bytes += lit_head_top * sizeof(literal);
for (l = 0; l < lit_head_top; l++) o, lmem[l].extra = 0;
cmem = (clause *) malloc(clauses * sizeof(clause));
if (¬cmem) {
  fprintf(stderr, "Oops, I can't allocate the cmem array!\n");
  exit(−13);
}
bytes += clauses * sizeof(clause);
```

See also section 82.

This code is used in section 40.

**53. Clearing the to-do stack.** To warm up, let's take care of the most basic operation, which simply assigns a forced value to a variable and propagates all the consequences until nothing more is obviously forced.

⟨ Clear the to-do stack 53 ⟩ ≡
```
while (to_do) {
  register uint c;
  k = to_do;
  o, to_do = vmem[k].link;
  if (vmem[k].status ≠ elim_quiet) {
    l = vmem[k].status ≡ forced_true ? pos_lit(k) : neg_lit(k);
    fprintf(erp_file, ""O"s"O".8s␣<-0\n", litname(l));
    o, vmem[k].stable = 1;
    ⟨ Delete all clauses that contain l 56 ⟩;
    ⟨ Delete bar(l) from all clauses 54 ⟩;
  }
  vars_gone++;
  if (sanity_checking) sanity();
}
if (mems > timeout) {
  if (verbose & show_basics) fprintf(stderr, "Timeout!\n");
  goto finish_up;    /* stick with the simplifications we've got so far */
}
```
This code is used in section 65.

**54.** ⟨ Delete bar(l) from all clauses 54 ⟩ ≡
```
for (o, ll = down(bar(l)); ¬is_lit(ll); o, ll = down(ll)) {
  o, c = mem[ll].cls;
  o, p = left(ll), q = right(ll);
  oo, right(p) = q, left(q) = p;
  free_cell(ll);    /* down(ll) unchanged */
  o, j = size(c) − 1;
  o, size(c) = j;
  if (j ≡ 1) {
    o, w = (p ≡ c ? mem[q].lit : mem[p].lit);
    if (verbose & show_details)
      fprintf(stderr, "clause␣"O"u␣reduces␣to␣"O"s"O".8s\n", c, litname(w));
    ⟨ Force literal w to be true 50 ⟩;
  }
  ⟨ Recompute clssig(c) 55 ⟩;
  if (o, slink(c) ≡ 0) o, slink(c) = strengthened, strengthened = c;
}
```
This code is used in section 53.

**55.** ⟨ Recompute clssig(c) 55 ⟩ ≡
```
{
  register ullng bits = 0;
  register uint t;
  for (o, t = right(c); ¬is_cls(t); o, t = right(t)) oo, bits |= litsig(mem[t].lit);
  o, clssig(c) = bits;
}
```
This code is used in section 54.

**56.** ⟨ Delete all clauses that contain $l$ 56 ⟩ ≡

```
for (o, ll = down(l); ¬is_lit(ll); o, ll = down(ll)) {
  o, c = mem[ll].cls;
  if (verbose & show_details)
    fprintf(stderr, "clause␣%u␣is␣satisfied␣by␣%s%.8s\n", c, litname(l));
  for (o, p = right(c); ¬is_cls(p); o, p = right(p))
    if (p ≠ ll) {
      o, w = mem[p].lit;
      o, q = up(p), r = down(p);
      oo, down(q) = r, up(r) = q;
      touch(w);
      oo, occurs(w)−−;
      if (occurs(w) ≡ 0) {
        if (verbose & show_details)
          fprintf(stderr, "literal␣%s%.8s␣no␣longer␣appears\n", litname(w));
        ⟨ Set literal w to false unless it's already set 51 ⟩;
      }
    }
  free_cells(right(c), left(c));
  o, size(c) = 0, clauses_gone++;
}
```

This code is used in section 53.

**57. Subsumption testing.** Our data structures make it fairly easy to find (and remove) all clauses that are subsumed by a given clause $C$, using an algorithm proposed by Armin Biere [*Lecture Notes in Computer Science* **3542** (2005), 59–70]: We choose a literal $l \in C$, then run through all clauses $C'$ that contain $l$. Most of the cases in which $C$ is not a subset of $C'$ can be ruled out quickly by looking at the sizes and signatures of $C$ and $C'$.

It would be nice to be able to go the other way, namely to start with a clause $C'$ and to determine whether or not it is subsumed by some $C$. That seems unfeasible; but there *is* a special case in which we do have some hope: When we resolve the clause $C_0 = x \vee \alpha$ with the clause $C_1 = \bar{x} \vee \beta$, to get $C' = \alpha \vee \beta$, we can assume that any clause $C$ contained in $C'$ contains an element of $\alpha \setminus \beta$ as well as an element of $\beta \setminus \alpha$; otherwise $C$ would subsume $C_0$ or $C_1$. Thus if $\alpha \setminus \beta$ and/or $\beta \setminus \alpha$ consists of a single element $l$, we can search through all clauses $C$ that contain $l$, essentially as above but with roles reversed.

(I wrote that last paragraph just in case it might come in useful some day; so far, this program only implements the idea in the *first* paragraph.)

```
⟨Remove clauses subsumed by c 57⟩ ≡
  if (verbose & show_subtrials) fprintf(stderr, "␣trying␣subsumption␣by␣"O"u\n", c);
  ⟨Choose a literal l ∈ c on which to branch 58⟩;
  ooo, s = size(c), bits = clssig(c), v = left(c);
  for (o, pp = down(l); ¬is_lit(pp); o, pp = down(pp)) {
    o, cc = mem[pp].cls;
    if (cc ≡ c) continue;
    sub_tries++;
    if (o, bits & ~clssig(cc)) continue;
    if (o, size(cc) < s) continue;
    ⟨If c is contained in cc, make l ≤ ll 59⟩;
    if (l > ll) sub_false++;
    else ⟨Remove the subsumed clause cc 60⟩;
  }
```

This code is used in sections 65 and 93.

**58.** Naturally we seek a literal that appears in the fewest clauses.

```
⟨Choose a literal l ∈ c on which to branch 58⟩ ≡
  ooo, p = right(c), l = mem[p].lit, k = occurs(l);
  for (o, p = right(p); ¬is_cls(p); o, p = right(p)) {
    o, ll = mem[p].lit;
    if (o, occurs(ll) < k) k = occurs(ll), l = ll;
  }
```

This code is used in section 57.

**59.** The algorithm here actually ends up with either $l < ll$ or $l > ll$ in all cases.

⟨ If $c$ is contained in $cc$, make $l \le ll$ 59 ⟩ ≡

```
o, q = v, qq = left(cc);
while (1) {
  oo, l = mem[q].lit, ll = mem[qq].lit;
  while (l < ll) {
    o, qq = left(qq);
    if (is_cls(qq)) ll = 0;
    else o, ll = mem[qq].lit;
  }
  if (l > ll) break;
  o, q = left(q);
  if (is_cls(q)) {
    l = 0; break;
  }
  o, qq = left(qq);
  if (is_cls(qq)) {
    ll = 0; break;
  }
}
```

This code is used in section 57.

**60.** ⟨ Remove the subsumed clause $cc$ 60 ⟩ ≡

```
{
  if (verbose & show_details) fprintf(stderr, "clause␣"O"u␣subsumes␣clause␣"O"u\n", c, cc);
  sub_total++;
  for (o, p = right(cc); ¬is_cls(p); o, p = right(p)) {
    o, q = up(p), r = down(p);
    oo, down(q) = r, up(r) = q;
    o, w = mem[p].lit;
    touch(w);
    oo, occurs(w)−−;
    if (occurs(w) ≡ 0) {
      if (verbose & show_details)
        fprintf(stderr, "literal␣"O"s"O".8s␣no␣longer␣appears\n", litname(q));
      ⟨ Set literal w to false unless it's already set 51 ⟩;
    }
  }
  free_cells(right(cc), left(cc));
  o, size(cc) = 0, clauses_gone++;
}
```

This code is used in section 57.

**61. Strengthening.** A similar algorithm can be used to find clauses $C'$ that, when resolved with a given clause $C$, become *stronger* (shorter). This happens when $C$ contains a literal $l$ such that $C$ would subsume $C'$ if $l$ were changed to $\bar{l}$ in $C$; then we can remove $\bar{l}$ from $C'$. [See Niklas Eén and Armin Biere, *Lecture Notes in Computer Science* **3569** (2005), 61–75.]

Thus I repeat the previous code, with the necessary changes for this modification. The literal called $l$ above is called $u$ in this program.

```
⟨Strengthen clauses that c can improve 61⟩ ≡
  {
    ooo, s = size(c), bits = clssig(c), v = left(c);
    for (o, vv = v; ¬is_cls(vv); o, vv = left(vv)) {
      register ullng ubits;
      o, u = mem[vv].lit;
      if (specialcase) ⟨Reject u unless it fills special conditions 95⟩;
      if (verbose & show_subtrials)
        fprintf(stderr, "␣trying␣to␣strengthen␣by␣"O"u␣and␣"O"s"O".8s\n", c, litname(u));
      o, ubits = bits & ∼litsig(u);
      for (o, pp = down(bar(u)); ¬is_lit(pp); o, pp = down(pp)) {
        str_tries++;
        o, cc = mem[pp].cls;
        if (o, ubits & ∼clssig(cc)) continue;
        if (o, size(cc) < s) continue;
        ⟨If c is contained in cc, except for u, make l ≤ ll 62⟩;
        if (l > ll) str_false++;
        else ⟨Remove bar(u) from cc 63⟩;
      }
    }
  }
```

This code is used in sections 65 and 94.

**62.**
```
⟨If c is contained in cc, except for u, make l ≤ ll 62⟩ ≡
  o, q = v, qq = left(cc);
  while (1) {
    oo, l = mem[q].lit, ll = mem[qq].lit;
    if (l ≡ u) l = bar(l);
    while (l < ll) {
      o, qq = left(qq);
      if (is_cls(qq)) ll = 0;
      else o, ll = mem[qq].lit;
    }
    if (l > ll) break;
    o, q = left(q);
    if (is_cls(q)) {
      l = 0; break;
    }
    o, qq = left(qq);
    if (is_cls(qq)) {
      ll = 0; break;
    }
  }
```

This code is used in section 61.

**63.** ⟨ Remove *bar*(*u*) from *cc* 63 ⟩ ≡

```
{
  register ullng ccbits = 0;
  if (verbose & show_details) fprintf(stderr,
        "clause␣"O"u␣loses␣literal␣"O"s"O".8s␣via␣clause␣"O"u\n", cc, litname(bar(u)), c);
  str_total++;
  for (o, p = right(cc); ; o, p = right(p)) {
    o, w = mem[p].lit;
    touch(w);
    if (w ≡ bar(u)) break;
    o, ccbits |= litsig(w);
  }
  oo, occurs(w)−−;
  if (occurs(w) ≡ 0) {
    if (verbose & show_details)
      fprintf(stderr, "literal␣"O"s"O".8s␣no␣longer␣appears\n", litname(w));
    ⟨ Set literal w to false unless it's already set 51 ⟩;
  }
  o, q = up(p), w = down(p);
  oo, down(q) = w, up(w) = q;
  o, q = right(p), w = left(p);
  oo, left(q) = w, right(w) = q;
  free_cell(p);
  for (p = q; ¬is_cls(p); o, p = right(p)) {
    o, q = mem[p].lit;
    touch(q);
    o, ccbits |= litsig(q);
  }
  o, clssig(cc) = ccbits;
  ⟨ Decrease size(cc) 64 ⟩;
  if (o, slink(cc) ≡ 0) o, slink(cc) = strengthened, strengthened = cc;
}
```

This code is used in section 61.

**64.** Clause *cc* shouldn't become empty at this point. For that could happen only if clause *c* had been a unit clause. (We don't use unit clauses for strengthening in such a baroque way; we handle them with the much simpler to-do list mechanism.)

⟨ Decrease *size*(*cc*) 64 ⟩ ≡

```
  oo, size(cc)−−;
  if (size(cc) ≤ 1) {
    if (size(cc) ≡ 0) confusion("strengthening");
    oo, w = mem[right(cc)].lit;
    if (verbose & show_details)
      fprintf(stderr, "clause␣"O"u␣reduces␣to␣"O"s"O".8s\n", cc, litname(w));
    ⟨ Force literal w to be true 50 ⟩;
  }
```

This code is used in section 63.

**65. Clearing the strengthened stack.** Whenever a clause gets shorter, it has new opportunities to subsume and/or strengthen other clauses. So we eagerly exploit all such opportunities.

```
⟨ Clear the strengthened stack 65 ⟩ ≡
  {
    register uint c;
    ⟨ Clear the to-do stack 53 ⟩;
    while (strengthened ≠ sentinel) {
      c = strengthened;
      o, strengthened = slink(c);
      if (o, size(c)) {
        o, slink(c) = 0;
        ⟨ Remove clauses subsumed by c 57 ⟩;
        ⟨ Clear the to-do stack 53 ⟩;
        if (o, size(c)) {
          specialcase = 0;
          ⟨ Strengthen clauses that c can improve 61 ⟩;
          ⟨ Clear the to-do stack 53 ⟩;
          o, clstime(c) = time;
          o, newsize(c) = 0;
        }
      }
    }
  }
```

This code is used in sections 83, 91, 93, and 94.

**66. Variable elimination.** The satisfiability problem is essentially the evaluation of the predicate $\exists x\,\exists y\, f(x,y)$, where $x$ is a variable and $y$ is a vector of other variables. Furthermore $f$ is expressed in conjunctive normal form (CNF); so we can write $f(x,y) = \bigl(x \lor \alpha(y)\bigr) \land \bigl(\bar{x} \lor \beta(y)\bigr) \land \gamma(y)$, where $\alpha$, $\beta$, and $\gamma$ are also in CNF. Since $\exists x\, f(x,y) = f(0,y) \lor f(1,y)$, we can eliminate $x$ and get the $x$-free problem $\exists y\,\bigl(\alpha(y) \lor \gamma(y)\bigr) \land \bigl(\beta(y) \lor \gamma(y)\bigr) = \exists y\,\bigl(\alpha(y) \lor \beta(y)\bigr) \land \gamma(y)$.

Computationally this means that we can replace all of the clauses that contain $x$ or $\bar{x}$ by the clauses of $\alpha(y) \lor \beta(y)$. And if $\alpha(y) = \alpha_1 \land \cdots \land \alpha_a$ and $\beta(y) = \beta_1 \land \cdots \land \beta_b$, those clauses are the so-called resolvents $(x \lor \alpha_i) \diamond (\bar{x} \lor \beta_j) = \alpha_i \lor \beta_j$, for $1 \le i \le a$ and $1 \le j \le b$.

Codewise, we want to compute the resolvent of $c$ with $cc$, given clauses $c$ and $cc$, assuming that $l$ and $ll = bar(l)$ are respectively contained in $c$ and $cc$.

The effect of the computation in this step will be to set $p = 0$ if the resolvent is a tautology (containing both $y$ and $\bar{y}$ for some $y$). Otherwise the cells of the resolvent will be $p$, ..., $left(left(1))$, $left(1)$. These cells will be linked together tentatively via their *left* links, thus not yet incorporated into the main data structures.

```
⟨ Resolve c and cc with respect to l 66 ⟩ ≡
  p = 1;
  oo, v = left(c), u = mem[v].lit;
  oo, vv = left(cc), uu = mem[vv].lit;
  while (u + uu) {
    if (u ≡ uu) ⟨ Copy u and move both v and vv left 72 ⟩
    else if (u ≡ bar(uu)) {
      if (u ≡ l) ⟨ Move both v and vv left 69 ⟩
      else ⟨ Return a tautology 73 ⟩;
    } else if (u > uu) ⟨ Copy u and move v left 70 ⟩
    else ⟨ Copy uu and move vv left 71 ⟩;
  }
```

This code is used in section 78.

**67.**
```
⟨ Move v left 67 ⟩ ≡
  {
    o, v = left(v);
    if (is_cls(v)) u = 0;
    else o, u = mem[v].lit;
  }
```

This code is used in sections 69 and 70.

**68.**
```
⟨ Move vv left 68 ⟩ ≡
  {
    o, vv = left(vv);
    if (is_cls(vv)) uu = 0;
    else o, uu = mem[vv].lit;
  }
```

This code is used in sections 69 and 71.

**69.**
```
⟨ Move both v and vv left 69 ⟩ ≡
  {
    ⟨ Move v left 67 ⟩;
    ⟨ Move vv left 68 ⟩;
  }
```

This code is used in sections 66 and 72.

**70.** ⟨ Copy $u$ and move $v$ left 70 ⟩ ≡

```
{
  q = p, p = get_cell();
  oo, left(q) = p, mem[p].lit = u;
  ⟨ Move v left 67 ⟩;
}
```

This code is used in section 66.

**71.** ⟨ Copy $uu$ and move $vv$ left 71 ⟩ ≡

```
{
  q = p, p = get_cell();
  oo, left(q) = p, mem[p].lit = uu;
  ⟨ Move vv left 68 ⟩;
}
```

This code is used in section 66.

**72.** ⟨ Copy $u$ and move both $v$ and $vv$ left 72 ⟩ ≡

```
{
  q = p, p = get_cell();
  oo, left(q) = p, mem[p].lit = u;
  ⟨ Move both v and vv left 69 ⟩;
}
```

This code is used in section 66.

**73.** ⟨ Return a tautology 73 ⟩ ≡

```
{
  if (p ≠ 1) o, free_cells(p, left(1));
  p = 0;
  break;
}
```

This code is used in section 66.

**74.** Eén and Biere, in their paper about preprocessing cited above, noticed that important simplifications are possible when $x$ is fully determined by other variables.

Formally we can try to partition the clauses $\alpha = \alpha^{(0)} \lor \alpha^{(1)}$ and $\beta = \beta^{(0)} \lor \beta^{(1)}$ in such a way that $(\alpha^{(0)} \land \beta^{(0)}) \lor (\alpha^{(1)} \land \beta^{(1)}) \le (\alpha^{(0)} \land \beta^{(1)}) \lor (\alpha^{(1)} \land \beta^{(0)})$; then we need not compute the resolvents $(\alpha^{(0)} \land \beta^{(0)})$ or $(\alpha^{(1)} \land \beta^{(1)})$, because the resolvents of "oppositely colored" $\alpha$'s and $\beta$'s imply all of the "same colored" ones. A necessary and sufficient condition for this to be possible is that the conditions $\alpha^{(0)} = \beta^{(0)} \ne \alpha^{(1)} = \beta^{(1)}$ are not simultaneously satisfiable.

For example, the desired condition holds if we can find a partition of the clauses such that $\alpha^{(0)} = \neg\beta^{(0)}$, because the clauses $\bigl(x \lor \neg\beta^{(0)}\bigr) \land \bigl(\bar{x} \lor \beta^{(0)}\bigr)$ imply that $x = \beta^{(0)}$ is functionally dependent on the other variables.

Another example is more trivial: We can clearly always take $\beta^{(0)} = \alpha^{(1)} = \emptyset$. Then the computation proceeds without any improvement. But this example shows that we can always assume that a suitable partitioning of the $\alpha$'s and $\beta$'s exists; hence the same program can drive the vertex elimination algorithm in either case.

The following program recognizes simple cases in which $\alpha^{(0)}$ consists of unit clauses $l_1 \land \cdots \land l_k$ and $\beta^{(0)}$ is a single clause $\bar{l}_1 \lor \cdots \lor \bar{l}_k$ equal to $\neg\alpha^{(0)}$. (Thus it detects a functional dependency that's AND, OR, NAND, or NOR.) If it finds such an example, it doesn't keep looking for another dependency, even though more efficient partitions may exist. It sets *beta0* $= cc$ when *cc* is the clause $\bar{x} \lor \bar{l}_1 \lor \cdots \lor \bar{l}_k$, and it sets *lmem*$[\bar{l}_i]$.*extra* $=$ *stamp* for $1 \le i \le k$; here *stamp* is an integer that uniquely identifies such literals. But if no such case is discovered, the program sets *beta0* $= 0$ and no literals have an *extra* that matches *stamp*.

(If I had more time I could look also for cases where $x = l_1 \oplus l_2$, or $x = \langle l_1 l_2 l_3 \rangle$, or $x = (l_1 ?\, l_2 {:}\, l_3)$, etc.)

⟨ Partition the $\alpha$'s and $\beta$'s if a simple functional dependency is found 74 ⟩ ≡

```
{
  register ullng stbits = 0;    /* signature of the l̄_i */
  beta0 = 0, stamp++;
  ll = bar(l);
  ⟨ Stamp all literals that appear with l in binary clauses 75 ⟩;
  if (stbits) {
    o, stbits |= litsig(ll);
    for (o, p = down(ll); ¬is_lit(p); o, p = down(p)) {
      o, c = mem[p].cls;
      if (o, (clssig(c) & ∼stbits) ≡ 0)
        ⟨ If the complements of all other literals in c are stamped, set beta0 = c and break 76 ⟩;
    }
  }
  if (beta0) {
    stamp++;
    ⟨ Stamp the literals of clause beta0 77 ⟩;
  }
}
```

This code is used in section 78.

**75.** ⟨Stamp all literals that appear with $l$ in binary clauses 75⟩ ≡

$\textbf{for}\ (o, p = down(l);\ \neg is\_lit(p);\ o, p = down(p))\ \{$
$\quad \textbf{if}\ (oo, size(mem[p].cls) \equiv 2)\ \{$
$\quad\quad o, q = right(p);$
$\quad\quad \textbf{if}\ (is\_cls(q))\ o, q = left(p);$
$\quad\quad oo, lmem[mem[q].lit].extra = stamp;$
$\quad\quad o, stbits \mathrel{|}= litsig(bar(mem[q].lit));$
$\quad \}$
$\}$

This code is used in section 74.

**76.** ⟨If the complements of all other literals in $c$ are stamped, set $beta0 = c$ and **break** 76⟩ ≡

$\{$
$\quad \textbf{for}\ (o, q = left(p);\ q \neq p;\ o, q = left(q))\ \{$
$\quad\quad \textbf{if}\ (is\_cls(q))\ \textbf{continue};$
$\quad\quad \textbf{if}\ (oo, lmem[bar(mem[q].lit)].extra \neq stamp)\ \textbf{break};$
$\quad \}$
$\quad \textbf{if}\ (q \equiv p)\ \{$
$\quad\quad beta0 = c;$
$\quad\quad \textbf{break};$
$\quad \}$
$\}$

This code is used in section 74.

**77.** ⟨Stamp the literals of clause $beta0$ 77⟩ ≡

$\textbf{if}\ (mem[p].cls \neq beta0 \lor mem[p].lit \neq ll)\ confusion(\texttt{"partitioning"});$
$\textbf{for}\ (o, q = left(p);\ q \neq p;\ o, q = left(q))\ \{$
$\quad \textbf{if}\ (is\_cls(q))\ \textbf{continue};$
$\quad oo, lmem[bar(mem[q].lit)].extra = stamp;$
$\}$

This code is used in section 74.

**78.** Now comes the main loop where we test whether the elimination of variable $x$ is desirable.

If both $x$ and *bar*($x$) occur in more than *cutoff* clauses, we don't attempt to do anything here, because we assume that the elimination of $x$ will almost surely add more clauses than it removes.

The resolvent clauses are formed as singly linked lists (via *left* fields), terminated by 0. They're linked together via *down* fields, starting at *down*(0) and ending at *last_new*.

⟨Either generate the clauses to eliminate variable $x$, or **goto** *elim_done* 78⟩ ≡

```
l = pos_lit(x);
oo, clauses_saved = occurs(l) + occurs(l + 1);
if ((occurs(l) > cutoff) ∧ (occurs(l + 1) > cutoff)) goto elim_done;
if ((ullng) occurs(l) ∗ occurs(l + 1) > occurs(l) + occurs(l + 1) + optimism) goto elim_done;
elim_tries++;
⟨Partition the α's and β's if a simple functional dependency is found 74⟩;
if (beta0 ≡ 0) {
  l++;    /∗ if at first you don't succeed, ... ∗/
  ⟨Partition the α's and β's if a simple functional dependency is found 74⟩;
}
if (beta0) func_total++;
if (verbose & show_restrials) {
  if (beta0)
    fprintf(stderr, "␣maybe␣elim␣"O"s␣("O"u,"O"d)\n", vmem[x].name.ch8, beta0, size(beta0) − 1);
  else fprintf(stderr, "␣maybe␣elim␣"O"s\n", vmem[x].name.ch8);
}
last_new = 0;
for (o, alf = down(l); ¬is_lit(alf); o, alf = down(alf)) {
  o, c = mem[alf].cls;
  ⟨Decide whether c belongs to α^(0) or α^(1) 79⟩;
  for (o, bet = down(ll); ¬is_lit(bet); o, bet = down(bet)) {
    o, cc = mem[bet].cls;
    if (cc ≡ beta0 ∧ alpha0) continue;
    if (cc ≠ beta0 ∧ ¬alpha0) continue;
    ⟨Resolve c and cc with respect to l 66⟩;
    if (p) {    /∗ we have a new resolvent ∗/
      o, left(p) = 0;    /∗ complete the tentative clause ∗/
      oo, down(last_new) = left(1);
      o, last_new = left(1), right(last_new) = p;
      if (−−clauses_saved < 0) ⟨Discard the new resolvents and goto elim_done 80⟩;
      up(last_new) = c, mem[last_new].cls = cc;    /∗ diagnostic only, no mem cost ∗/
    }
  }
}
o, down(last_new) = 0;    /∗ complete the vertical list of new clauses ∗/
```

This code is used in section 83.

**79.** ⟨ Decide whether $c$ belongs to $\alpha^{(0)}$ or $\alpha^{(1)}$ 79 ⟩ ≡

```
if (beta0 ≡ 0) alpha0 = 1;
else {
  alpha0 = 0;
  if (o, size(c) ≡ 2) {
    o, q = right(c);
    if (q ≡ alf) q = left(c);
    if (oo, lmem[mem[q].lit].extra ≡ stamp) alpha0 = 1;    /* yes, c ∈ α^(0) */
  }
}
```

This code is used in section 78.

**80.** Too bad: We found more resolvents than the clauses they would replace.

⟨ Discard the new resolvents and **goto** *elim_done* 80 ⟩ ≡

```
{
  for (o, p = down(0); ; o, p = down(p)) {
    o, free_cells(right(p), p);
    if (p ≡ last_new) break;
  }
  goto elim_done;
}
```

This code is used in section 78.

**81.** The *stamp* won't overflow because I'm not going to increase it $2^{64}$ times. (Readers in the 22nd century might not believe me though, if Moore's Law continues.)

⟨ Global variables 4 ⟩ +≡

```
ullng stamp;    /* a time stamp for unique identification */
uint beta0;    /* a clause that defines β^(0) in a good partition */
uint alpha0;    /* set to 1 if c is part of α^(0) */
uint last_new;    /* the beginning of the last newly resolved clause */
uint alf, bet;    /* loop indices for α_i and β_j */
int clauses_saved;    /* eliminating x saves at most this many clauses */
uint *bucket;    /* heads of lists of candidates for elimination */
```

**82.** ⟨ Allocate the subsidiary arrays 52 ⟩ +≡

```
if (buckets < 2) buckets = 2;
bucket = (uint *) malloc((buckets + 1) * sizeof(uint));
if (¬bucket) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣bucket␣array!\n");
  exit(−14);
}
bytes += (buckets + 1) * sizeof(uint);
```

**83.** ⟨ Try to eliminate variables 83 ⟩ ≡

```
⟨ Place candidates for elimination into buckets 84 ⟩;
for (b = 2; b ≤ buckets; b++)
  if (o, bucket[b]) {
    for (x = bucket[b]; x; o, x = vmem[x].blink)
      if (o, vmem[x].stable ≡ 0) {
        if (sanity_checking) sanity();
        ⟨ Either generate the clauses to eliminate variable x, or goto elim_done 78 ⟩;
        ⟨ Eliminate variable x, replacing its clauses by the new resolvents 85 ⟩;
        if (sanity_checking) sanity();
        ⟨ Clear the strengthened stack 65 ⟩;
      elim_done: o, vmem[x].stable = 1;
      }
  }
```

This code is used in section 91.

**84.** ⟨ Place candidates for elimination into buckets 84 ⟩ ≡

```
for (b = 2; b ≤ buckets; b++) o, bucket[b] = 0;
for (x = vars; x; x−−) {
  if (o, vmem[x].stable) continue;
  if (vmem[x].status) confusion("touched␣and␣eliminated");
  l = pos_lit(x);
  oo, p = occurs(l), q = occurs(l + 1);
  if (p > cutoff ∧ q > cutoff) goto reject;
  b = p + q;
  if ((ullng) p * q > b + optimism) goto reject;
  if (b > buckets) b = buckets;
  oo, vmem[x].blink = bucket[b];
  o, bucket[b] = x; continue;
reject: o, vmem[x].stable = 1;
}
```

This code is used in section 83.

**85.** ⟨ Eliminate variable $x$, replacing its clauses by the new resolvents 85 ⟩ ≡

```
if (verbose & show_details) {
  fprintf(stderr, "elimination␣of␣"O"s", vmem[x].name.ch8);
  if (beta0) fprintf(stderr, "␣("O"u,"O"d)", beta0, size(beta0) − 1);
  fprintf(stderr, "␣saves␣"O"d␣clause"O"s\n", clauses_saved, clauses_saved ≡ 1 ? "" : "s");
}
if (verbose & show_resolutions) print_clauses_for(pos_lit(x)), print_clauses_for(neg_lit(x));
⟨ Update the erp file for the elimination of x 86 ⟩;
oo, down(last_new) = 0, last_new = down(0);
v = pos_lit(x);
⟨ Replace the clauses of v by new resolvents 87 ⟩;
v++;
⟨ Replace the clauses of v by new resolvents 87 ⟩;
⟨ Recycle the cells of clauses that involve v 88 ⟩;
v−−;
⟨ Recycle the cells of clauses that involve v 88 ⟩;
o, vmem[x].status = elim_res, vars_gone++;
clauses_gone += clauses_saved;
```

This code is used in section 83.

**86.** ⟨ Update the `erp` file for the elimination of $x$ 86 ⟩ ≡

```
if (beta0) {
  fprintf(erp_file, ""O"s"O".8s␣<-1\n", litname(l));
  for (o, q = right(beta0); ¬is_cls(q); o, q = right(q))
    if (o, mem[q].lit ≠ ll) fprintf(erp_file, "␣"O"s"O".8s", litname(mem[q].lit));
  fprintf(erp_file, "\n");
} else {
  o, k = occurs(l), v = l;
  if (o, k > occurs(ll)) k = occurs(ll), v = ll;
  fprintf(erp_file, ""O"s"O".8s␣<-"O"d\n", litname(bar(v)), k);
  for (o, p = down(v); ¬is_lit(p); o, p = down(p)) {
    for (o, q = right(p); q ≠ p; o, q = right(q))
      if (¬is_cls(q)) o, fprintf(erp_file, "␣"O"s"O".8s", litname(mem[q].lit));
    fprintf(erp_file, "\n");
  }
}
```

This code is used in section 85.

**87.** We can't remove the old cells until *after* inserting the new ones, because we don't want false claims of pure literals. But we *can* safely detach those cells from the old clause heads.

⟨ Replace the clauses of $v$ by new resolvents 87 ⟩ ≡

```
for (o, p = down(v); ¬is_lit(p); o, p = down(p)) {
  o, c = mem[p].cls;
  o, q = right(c), r = left(c);
  oo, left(q) = r, right(r) = q;
  ⟨ Replace clause c by a new resolvent, if any 89 ⟩;
}
```

This code is used in section 85.

**88.** Every literal that appears in a new resolvent will be touched when we recycle the clauses that were resolved.

⟨ Recycle the cells of clauses that involve $v$ 88 ⟩ ≡

```
for (o, p = down(v); ¬is_lit(p); o, p = down(p)) {
  for (o, q = right(p); q ≠ p; o, q = right(q)) {
    o, r = up(q), w = down(q);
    oo, down(r) = w, up(w) = r;
    o, w = mem[q].lit;
    touch(w);
    oo, occurs(w)−−, littime(w) = time;
    if (occurs(w) ≡ 0) {
      if (verbose & show_details)
        fprintf(stderr, "literal␣"O"s"O".8s␣no␣longer␣appears\n", litname(w));
      ⟨ Set literal w to false unless it's already set 51 ⟩;
    }
  }
  free_cells(right(p), p);
}
```

This code is used in section 85.

**89.** A new resolvent *last_new* is waiting to be launched as an official clause, unless *last_new* = 0.

⟨ Replace clause $c$ by a new resolvent, if any 89 ⟩ ≡

```
if (last_new) {
  if (verbose & show_details)
    fprintf(stderr, "clause␣"O"u␣now␣"O"u␣res␣"O"u\n", c, up(last_new), mem[last_new].cls);
  o, pp = down(last_new);
  ⟨ Install last_new into position c 90 ⟩;
  if (verbose & show_resolutions) print_clause(c);
  o, newsize(c) = 1;
  o, last_new = pp;
}
else o, size(c) = 0;
```

This code is used in section 87.

**90.** ⟨ Install *last_new* into position $c$ 90 ⟩ ≡

```
for (q = last_new, r = c, s = 0, bits = 0; q; o, r = q, q = left(q)) {
  o, u = mem[q].lit;
  oo, occurs(u)++;
  o, w = up(u);
  oo, up(u) = down(w) = q;
  o, up(q) = w, down(q) = u;
  o, bits |= litsig(u);
  o, right(q) = r;
  o, mem[q].cls = c;
  s++;
}
oo, size(c) = s, clssig(c) = bits;
oo, left(c) = last_new, right(c) = r, left(r) = c;
if (s ≡ 1) {
  o, w = mem[r].lit;
  if (verbose & show_details) fprintf(stderr, "clause␣"O"u␣is␣just␣"O"s"O".8s\n", c, litname(w));
  ⟨ Force literal w to be true 50 ⟩;
}
```

This code is used in section 89.

**91. The dénouement.** (*dénouement*, n.: The final resolution of the intricacies of a plot; the outcome or resolution of a doubtful series of occurrences.)

⟨ Preprocess until everything is stable 91 ⟩ ≡
```
if (verbose & show_initial_clauses) print_all();
if (sanity_checking) sanity();
⟨ Put all clauses into the strengthened stack 92 ⟩;
⟨ Clear the strengthened stack 65 ⟩;
for (time = 1; time ≤ maxrounds; time++) {
  int progress = vars_gone;
  if (verbose & show_rounds)
    fprintf(stderr, "beginning round %u (%d vars, %d clauses gone, %llu mems)\n",
        time, vars_gone, clauses_gone, mems);
  ⟨ Try to eliminate variables 83 ⟩;
  if (progress ≡ vars_gone ∨ vars_gone ≡ vars) break;
  ⟨ Do a round of subsumption/strengthening on the new clauses 93 ⟩;
}
if (time > maxrounds) time = maxrounds;
```
This code is used in section 3.

**92.** At the beginning we might as well consider every clause to be "strengthened," because we want to exploit its ability to subsume and strengthen other clauses.

⟨ Put all clauses into the strengthened stack 92 ⟩ ≡
```
o, slink(lit_head_top) = sentinel, newsize(lit_head_top) = 0;
for (c = lit_head_top + 1; is_cls(c); c++) o, slink(c) = c − 1, newsize(c) = 0;
strengthened = c − 1;
```
This code is used in section 91.

**93.** Clauses that have been strengthened have also been fully exploited at this point. But the other existing clauses might subsume any of the new clauses generated by the last round of variable elimination, if all of their literals appear in at least one new clause. Such a clause $C$ might also strengthen another new clause $C'$, if $C$ itself is new, or if all but one of $C$'s literals are in $C'$ and so is the complement of the other.

The value of *newsize*$(c)$ at this point is 1 if and only if $c$ is new, otherwise it's 0. (At least, this statement is true whenever *size*$(c)$ is nonzero. All clauses with *size*$(c) = 0$ are permanently gone and essentially forgotten.)

Also, a given literal $l$ has appeared in a new clause of the current round if and only if *littime*$(l) =$ *time*.

So we run through all such literals, adding 4 to *newsize*$(c)$ for each clause they're in, also ORing 2 into *newsize*$(c)$ for each clause that their complement is in. The resulting *newsize* values will help us to decide a reasonably high speed whether an existing clause can be exploited.

```
⟨ Do a round of subsumption/strengthening on the new clauses 93 ⟩ ≡
  for (l = 2; is_lit(l); l++) {
    if ((l & 1) ≡ 0 ∧ (o, vmem[thevar(l)].status)) {
      l++; continue;   /* bypass eliminated variables */
    }
    if (o, littime(l) ≡ time) ⟨ Update newsize info for l's clauses 96 ⟩;
  }
  for (c = lit_head_top; is_cls(c); c++)
    if (o, size(c)) {
      if (clstime(c) < time) {   /* c not recently exploited */
        if (o, size(c) ≡ newsize(c) ≫ 2) {
          ⟨ Remove clauses subsumed by c 57 ⟩;
          ⟨ Clear the strengthened stack 65 ⟩;
        } else if (newsize(c) & 1) confusion("new␣clause␣not␣all␣new");
        if (newsize(c) & #3) ⟨ Maybe try to strengthen with c 94 ⟩;
      }
      o, newsize(c) = 0;
    }
```

This code is used in section 91.

**94.**
```
⟨ Maybe try to strengthen with c 94 ⟩ ≡
  {
    if (newsize(c) & 1) specialcase = 0;   /* c is a new clause */
    else {
      if (newsize(c) ≫ 2 < size(c) − 1) specialcase = −1;
      else specialcase = 1;
    }
    if (specialcase ≥ 0) {
      ⟨ Strengthen clauses that c can improve 61 ⟩;
      ⟨ Clear the strengthened stack 65 ⟩;
    }
  }
```

This code is used in section 93.

**95.**
```
⟨ Reject u unless it fills special conditions 95 ⟩ ≡
  {
    if (o, littime(bar(u)) ≠ time) continue;   /* reject if ū not new */
    if (o, newsize(c) ≫ 2 ≠ size(c) − (littime(u) ≠ time)) continue;
        /* reject if all other literals of c aren't new */
  }
```

This code is used in section 61.

**96.** ⟨ Update *newsize* info for *l*'s clauses 96 ⟩ ≡

```
{
  for (o, p = down(l); ¬is_lit(p); o, p = down(p)) {
    o, c = mem[p].cls;
    oo, newsize(c) += 4;
  }
  for (o, p = down(bar(l)); ¬is_lit(p); o, p = down(p)) {
    o, c = mem[p].cls;
    oo, newsize(c) |= 2;
  }
}
```

This code is used in section 93.

**97.** ⟨ Output the simplified clauses 97 ⟩ ≡

```
for (c = lit_head_top; is_cls(c); c++)
  if (o, size(c)) {
    for (o, p = right(c); ¬is_cls(p); o, p = right(p)) {
      o, l = mem[p].lit;
      printf(" "O"s"O".8s", litname(l));
    }
    printf("\n");
  }
if (vars_gone ≡ vars) {
  if (clauses_gone ≠ clauses) confusion("vars gone but not clauses");
  if (verbose & show_basics) fprintf(stderr, "No clauses remain.\n");
} else if (clauses_gone ≡ clauses) confusion("clauses gone but not vars");
else if (verbose & show_basics) fprintf(stderr,
      ""O"d variable"O"s and "O"d clause"O"s removed ("O"d round"O"s).\n", vars_gone,
      vars_gone ≡ 1 ? "" : "s", clauses_gone, clauses_gone ≡ 1 ? "" : "s", time, time ≡ 1 ? "" : "s");
if (0) {
unsat: fprintf(stderr, "The clauses are unsatisfiable.\n");
}
```

This code is used in section 3.

**98.** ⟨ Subroutines 26 ⟩ +≡

```
void confusion(char *id)
{   /* an assertion has failed */
  fprintf(stderr, "This can't happen ("O"s)!\n", id);
  exit(−69);
}
void debugstop(int foo)
{   /* can be inserted as a special breakpoint */
  fprintf(stderr, "You rang("O"d)?\n", foo);
}
```

**99. Index.**

# SAT12