June 27, 2011 at 03:35

1. Intro. This program is the fifteenth in a series of exploratory studies by which I'm attempting to gain first-hand experience with BDD structures, as I prepare Section 7.1.4 of *The Art of Computer Programming*. It's based on BDD14, but it does everything with ZDDs instead of BDDs.

In this program I try to implement simplified versions of the basic routines that are needed in a "large" ZDD package.

The computation is governed by primitive commands in a language called ZDDL; these commands can either be read from a file or typed online (or both). ZDDL commands have the following simple syntax, where ⟨ number ⟩ denotes a nonnegative decimal integer:

⟨ const ⟩ ← `c0` | `c1` | `c2`
⟨ var ⟩ ← `x`⟨ number ⟩ | `e`⟨ number ⟩
⟨ fam ⟩ ← `f`⟨ number ⟩
⟨ atom ⟩ ← ⟨ const ⟩ | ⟨ var ⟩ | ⟨ fam ⟩
⟨ expr ⟩ ← ⟨ unop ⟩⟨ atom ⟩ | ⟨ atom ⟩⟨ binop ⟩⟨ atom ⟩ |
⟨ atom ⟩⟨ ternop ⟩⟨ atom ⟩⟨ ternop ⟩⟨ atom ⟩
⟨ command ⟩ ← ⟨ special ⟩ | ⟨ fam ⟩`=`⟨ expr ⟩ | ⟨ fam ⟩`=.`

[Several operations appropriate for Boolean functions, such as quantification and functional composition, were implemented in BDD14, but they are omitted here; on the other hand, several new operations, appropriate for families of subsets, are now present. The constants `c0`, `c1`, and `c2` are what TAOCP calls $\emptyset$, $\wp$, and ϵ. The special commands ⟨ special ⟩, the unary operators ⟨ unop ⟩, the binary operators ⟨ binop ⟩, and the ternary operators ⟨ ternop ⟩ are explained below. One short example will give the general flavor: After the commands

```
x4
f1=x1&x2
f2=e3|c2
f3=~f1
f4=f3^f2
```

four families of subsets of $\{e_0,\ldots,e_4\}$ are present: Family f_1 consists of all eight subsets that contain both e_1 and e_2; f_2 is the family of two subsets, $\{e_3\}$ and $\emptyset$; f_3 is the family of all subsets do not contain both e_1 and e_2; f_4 is the family of all subsets that are in f_3 and not in f_2, or vice versa; since f_2 is contained in f_3, `f4=f3>f2` would give the same result in this case.. (We could also have defined f_3 with `f3=c1^f1`, because `c1` stands for the family of *all* subsets. Note the distinction between e_j and x_j: The former is an element, or the family consisting of a single one-element set; the latter is the family consisting of all sets containing element e_j.) A subsequent command '`f1=.`' will undefine f_1.

The first command in this example specifies that `x4` will be the largest `x` variable. (We insist that the variables of all ZDDs belong to a definite, fixed set; this restriction greatly simplifies the program logic.)

If the command line specifies an input file, all commands are taken from that file and standard input is ignored. Otherwise the user is prompted for commands.

2. For simplicity, I do my own memory allocation in a big array called *mem*. The bottom part of that array is devoted to ZDD nodes, which each occupy two octabytes. The upper part is divided into dynamically allocated pages of a fixed size (usually 4096 bytes). The cache of computed results, and the hash tables for each variable, are kept in arrays whose elements appear in the upper pages. These elements need not be consecutive, because the kth byte of each dynamic array is kept in location $mem[b[k \gg 12] + (k \,\&\, \#\mathtt{fff})]$, for some array b of base addresses.

Each node of the ZDD base is responsible for roughly 28 bytes in *mem*, assuming 16 bytes for the node itself, plus about 8 for its entry in a hash table, plus about 4 for its entry in a cache. (I could reduce the storage cost from 28 to 21 by choosing algorithms that run slower; but I decided to give up some space in the interests of time. For example, I'm devoting four bytes to each reference count, so that there's no need to consider saturation. And this program uses linear probing for its hash tables, at the expense of about 3 bytes per node, because I like the sequential memory accesses of linear probing.)

Many compile-time parameters affect the sizes of various tables and the heuristic strategies of various methods adopted here. To browse through them all, see the entry "Tweakable parameters" in the index at the end.

3. Here's the overall program structure:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "gb_flip.h"     /* random number generator */
#define verbose Verbose     /* because 'verbose' is long in libgb */
  ⟨Type definitions 11⟩
  ⟨Global variables 5⟩
  ⟨Templates for subroutines 26⟩
  ⟨Subroutines 8⟩
  main(int argc, char *argv[])
  {
    ⟨Local variables 19⟩;
    ⟨Check the command line 4⟩;
    ⟨Initialize everything 6⟩;
    while (1) ⟨Read a command and obey it; goto alldone if done 108⟩;
  alldone: ⟨Print statistics about this run 7⟩;
    exit(0);     /* normal termination */
  }
```

4.
```
#define file_given (argc ≡ 2)
⟨Check the command line 4⟩ ≡
  if (argc > 2 ∨ (file_given ∧ ¬(infile = fopen(argv[1], "r")))) {
    fprintf(stderr, "Usage:␣%s␣[commandfile]\n", argv[0]);
    exit(−1);
  }
```

This code is used in section 3.

5.
```
⟨Global variables 5⟩ ≡
  FILE *infile;     /* input file containing commands */
  int verbose = −1;     /* master control for debugging output; −1 gives all */
```

See also sections 9, 14, 22, 31, 41, 43, 52, 60, 110, 130, 134, 152, 155, and 158.

This code is used in section 3.

6. ⟨ Initialize everything 6 ⟩ ≡

```
gb_init_rand(0);    /* initialize the random number generator */
```

See also sections 10, 13, 45, and 63.

This code is used in section 3.

7. One of the main things I hope to learn with this program is the total number of *mems* that the computation needs, namely the total number of memory references to octabytes.

I'm not sure how many mems to charge for recursion overhead. A machine like MMIX needs to use memory only when the depth gets sufficiently deep that 256 registers aren't enough; then it needs two mems for each saved item (one to push it and another to pop it). Most of MMIX's recursive activity takes place in the deepest levels, whose parameters never need to descend to memory. So I'm making a separate count of *rmems*, the number of entries to recursive subroutines.

Some of the mems are classified as *zmems*, because they arise only when zeroing out pages of memory during initializations.

```
#define o mems++    /* a convenient macro for instrumenting a memory access */
#define oo mems += 2
#define ooo mems += 3
#define oooo mems += 4
#define rfactor 4.0    /* guesstimate used for weighted mems in TAOCP */
#define zfactor 1.0    /* guesstimate used for weighted mems in TAOCP */
```

⟨ Print statistics about this run 7 ⟩ ≡

```
printf("Job␣stats:\n");
printf("␣␣%llu␣mems␣plus␣%llu␣rmems␣plus␣%llu␣zmems␣(%.4g)\n", mems, rmems, zmems,
    mems + rfactor * rmems + zfactor * zmems);
⟨ Print total memory usage 18 ⟩;
```

This code is used in sections 3 and 156.

8. ⟨ Subroutines 8 ⟩ ≡

```
void show_stats(void)
{
  printf("stats:␣%d/%d␣nodes,␣%d␣dead,␣%d␣pages,", totalnodes, nodeptr − botsink, deadnodes,
      topofmem − pageptr);
  printf("␣%llu␣mems,␣%llu␣rmems,␣%llu␣zmems,␣%.4g\n", mems, rmems, zmems,
      mems + rfactor * rmems + zfactor * zmems);
}
```

See also sections 15, 16, 17, 23, 25, 27, 36, 42, 44, 46, 47, 50, 53, 54, 55, 56, 57, 59, 68, 72, 73, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 97, 99, 101, 103, 104, 105, 107, 120, 129, 135, 136, 139, 143, 146, 149, 153, 156, and 159.

This code is used in section 3.

9. This program uses '**long long**' to refer to 64-bit integers, because a single '**long**' isn't treated consistently by the C compilers available to me. (When I first learned C, '**int**' was traditionally '**short**', so I was obliged to say '**long**' when I wanted 32-bit integers. Consequently the programs of the Stanford GraphBase, written in the 90s, now get 64-bit integers—contrary to my original intent. C'est tragique; c'est la vie.)

⟨ Global variables 5 ⟩ +≡

```
unsigned long long mems, rmems, zmems;    /* mem counters */
```

10. ⟨ Initialize everything 6 ⟩ +≡

```
if (sizeof(long long) ≠ 8) {
  fprintf(stderr, "Sorry,␣I␣assume␣that␣sizeof(long␣long)␣is␣8!\n");
  exit(−2);
}
```

11. Speaking of compilers, the one I use at present insists that pointers occupy 64 bits. As a result, I need to pack and unpack pointer data, in all the key data structures of this program; otherwise I would basically be giving up half of my memory and half of the hardware cache.

I could solve this problem by using arrays with integer subscripts. Indeed, that approach would be simple and clean.

But I anticipate doing some fairly long calculations, and speed is also important to me. So I've chosen a slightly more complex (and slightly dirtier) approach, equivalent to using short pointers; I wrap such pointers up with syntax that doesn't offend my compiler. The use of this scheme allows me to use the convenient syntax of C for fields within structures.

Namely, data is stored here with a type called *addr*, which is simply an unsigned 32-bit integer. An *addr* contains all the information of a pointer, since I'm not planning to use this program with more than 2^{32} bytes of memory. It has a special name only to indicate its pointerly nature.

With this approach the program goes fast, as with usual pointers, because it doesn't have to shift left by 4 bits and add the base address of *mem* whenever addressing the memory. But I do limit myself to ZDD bases of at most about 30 million nodes.

(At the cost of shift-left-four each time, I could extend this scheme to handling a 35-bit address space, if I ever get a computer with 32 gigabytes of RAM. I still would want to keep 32-bit pointers in memory, in order to double the effective cache size.)

The *addr_* macro converts an arbitrary pointer to an *addr*.

#define *addr_*(*p*) ((*addr*)(**size_t**)(*p*))

⟨ Type definitions 11 ⟩ ≡
  typedef unsigned int addr;

See also sections 12, 20, and 40.

This code is used in section 3.

12. Dynamic arrays. Before I get into the ZDD stuff, I might as well give myself some infrastructure to work with.

The giant *mem* array mentioned earlier has nodes at the bottom, in locations *mem* through $nodeptr - 1$. It has pages at the top, in locations *pageptr* through $mem + memsize - 1$. We must therefore keep $nodeptr \le pageptr$.

A node has four fields, called *lo*, *hi*, *xref*, and *index*. I shall explain their significance eventually, when I *do* "get into the ZDD stuff."

A page is basically unstructured, although we will eventually fill it either with hash-table data or cache memos.

The *node_* and *page_* macros are provided to make pointers from stored items of type **addr**.

```
#define logpagesize 12   /* must be at least 4 */
#define memsize (1 << 29)   /* bytes in mem, must be a multiple of pagesize */
#define pagesize (1 << logpagesize)   /* the number of bytes per page */
#define pagemask (pagesize - 1)
#define pageints (pagesize/sizeof(int))
#define node_(a) ((node *)(size_t)(a))
#define page_(a) ((page *)(size_t)(a))
```

⟨Type definitions 11⟩ +≡

```
  typedef struct node_struct {
    addr lo, hi;
    int xref;   /* reference count minus one */
    unsigned int index;   /* variable ID followed by random bits */
  } node;
  typedef struct page_struct {
    addr dat[pageints];
  } page;
```

13. Here's how we launch the dynamic memory setup.

Incidentally, I tried to initialize *mem* by declaring it to be a variable of type **void** ∗, then saying '$mem = malloc(memsize)$'. But that failed spectacularly, because the geniuses who developed the standard library for my 64-bit version of Linux decided in their great wisdom to make *malloc* return a huge pointer like #`2adaf3739010`, even when the program could fit comfortably in a 30-bit address space. D'oh.

```
#define topofmem ((page *) &mem[memsize])
```

⟨Initialize everything 6⟩ +≡

```
  botsink = (node *) mem;   /* this is the sink node for the all-zero function */
  topsink = botsink + 1;   /* this is the sink node for the all-one function */
  o, botsink→lo = botsink→hi = addr_(botsink);
  o, topsink→lo = topsink→hi = addr_(topsink);
  oo, botsink→xref = topsink→xref = 0;
  totalnodes = 2;
  nodeptr = topsink + 1;
  pageptr = topofmem;
```

14. ⟨ Global variables 5 ⟩ +≡

```
char mem[memsize]; /* where we store most of the stuff */
node *nodeptr; /* the smallest unused node in mem */
page *pageptr; /* the smallest used page in mem */
node *nodeavail; /* stack of nodes available for reuse */
page *pageavail; /* stack of pages available for reuse */
node *botsink, *topsink; /* the sink nodes, which never go away */
int totalnodes; /* this many nodes are currently in use */
int deadnodes; /* and this many of them currently have xref < 0 */
int leasesonlife = 1; /* times to delay before giving up */
```

15. Here's how we get a fresh (but uninitialized) node. The *nodeavail* stack is linked by its *xref* fields.

If memory is completely full, Λ is returned. In such cases we need not abandon all hope; a garbage collection may be able to reclaim enough memory to continue. (I've tried to write this entire program in such a way that such temporary failures are harmless.)

⟨ Subroutines 8 ⟩ +≡

```
node *reserve_node(void)
{
  register node *r = nodeavail;
  if (r) o, nodeavail = node_(nodeavail→xref);
  else {
    r = nodeptr;
    if (r < (node *) pageptr) nodeptr++;
    else {
      leasesonlife--;
      fprintf(stderr, "NULL␣node␣forced␣(%d␣pages,␣%d␣nodes,␣%d␣dead)\n", topofmem − pageptr,
          nodeptr − botsink, deadnodes);
      fprintf(stderr, "(I␣will␣try␣%d␣more␣times)\n", leasesonlife);
      if (leasesonlife ≡ 0) {
        show_stats(); exit(−98); /* sigh */
      }
      return Λ;
    }
  }
  totalnodes++;
  return r;
}
```

16. Conversely, nodes can always be recycled. In such cases, there had better not be any other nodes pointing to them.

⟨ Subroutines 8 ⟩ +≡

```
void free_node(register node *p)
{
  o, p→xref = addr_(nodeavail);
  nodeavail = p;
  totalnodes--;
}
```

17. Occupation and liberation of pages is similar, but it takes place at the top of *mem*.

⟨ Subroutines 8 ⟩ +≡

```
page *reserve_page(void)
{
  register page *r = pageavail;
  if (r) o, pageavail = page_(pageavail→dat[0]);
  else {
    r = pageptr − 1;
    if ((node *) r ≥ nodeptr) pageptr = r;
    else {
      leasesonlife −−;
      fprintf(stderr, "NULL␣page␣forced␣(%d␣pages,␣%d␣nodes,␣%d␣dead)\n", topofmem − pageptr,
           nodeptr − botsink, deadnodes);
      fprintf(stderr, "(I␣will␣try␣%d␣more␣times)\n", leasesonlife);
      if (leasesonlife ≡ 0) {
        show_stats(); exit(−97);    /* sigh */
      }
      return Λ;
    }
  }
  return r;
}
void free_page(register page *p)
{
  o, p→dat[0] = addr_(pageavail);
  pageavail = p;
}
```

18. ⟨ Print total memory usage 18 ⟩ ≡

```
j = nodeptr − (node *) mem;
k = topofmem − pageptr;
printf("␣␣%llu␣bytes␣of␣memory␣(%d␣nodes,␣%d␣pages)\n", ((long long) j) * sizeof(node) + ((long
     long) k) * sizeof(page), j, k);
```

This code is used in section 7.

19. ⟨ Local variables 19 ⟩ ≡

```
register int j, k;
```

See also section 113.

This code is used in section 3.

20. Variables and hash tables. Our ZDD base represents functions on the variables x_v for $0 \le v < \mathit{varsize} - 1$, where *varsize* is a power of 2.

When x_v is first mentioned, we create a *var* record for it, from which it is possible to find all the nodes that branch on this variable. The list of all such nodes is implicitly present in a hash table, which contains a pointer to node (v, l, h) near the hash address of the pair (l, h). This hash table is called the *unique table* for v, because of the ZDD property that no two nodes have the same triple of values (v, l, h).

When there are n nodes that branch on x_v, the unique table for v has size m, where m is a power of 2 such that n lies between $m/8$ and $3m/4$, inclusive. Thus at least one of every eight table slots is occupied, and at least one of every four is unoccupied, on the average. If $n = 25$, for example, we might have $m = 64$ or $m = 128$; but $m = 256$ would make the table too sparse.

Each unique table has a maximum size, which must be small enough that we don't need too many base addresses for its pages, yet large enough that we can accommodate big ZDDs. If, for example, $\mathit{logmaxhashsize} = 19$ and $\mathit{logpagesize} = 12$, a unique table might contain as many as 2^{19} **addr**s, filling 2^9 pages. Then we must make room for 512 base addresses in each *var* record, and we can handle up to $2^{19} - 2^{17} = 393216$ nodes that branch on any particular variable.

```
#define logmaxhashsize 21
#define slotsperpage (pagesize/sizeof(addr))
#define maxhashpages (((1 << logmaxhashsize) + slotsperpage - 1)/slotsperpage)
```

⟨ Type definitions 11 ⟩ +≡

```
typedef struct var_struct {
  addr proj;   /* address of the projection function x_v */
  addr taut;   /* address of the function x̄_1 ... x̄_{v-1} */
  addr elt;   /* address of x_v ∧ S_1(x_1, ..., x_n) */
  int free;   /* the number of unused slots in the unique table for v */
  int mask;   /* the number of slots in that unique table, times 4, minus 1 */
  addr base[maxhashpages];   /* base addresses for its pages */
  int name;   /* the user's name (subscript) for this variable */
  int aux;   /* flag used by the sifting algorithm */
  struct var_struct *up, *down;   /* the neighboring active variables */
} var;
```

21. Every node p that branches on x_v in the ZDD has a field *p→index*, whose leftmost *logvarsize* bits contain the index v. The rightmost $32 - \mathit{logvarsize}$ bits of *p→index* are chosen randomly, in order to provide convenient hash coding.

The SGB random-number generator used here makes four memory references per number generated.

N.B.: The hashing scheme will fail dramatically unless $\mathit{logvarsize} + \mathit{logmaxhashsize} \le 32$.

```
#define logvarsize 10
#define varsize (1 << logvarsize)   /* the number of permissible variables */
#define varpart(x) ((x) >> (32 - logvarsize))
#define initnewnode(p, v, l, h) oo, p->lo = addr_(l), p->hi = addr_(h), p->xref = 0,
          oooo, p->index = ((v) << (32 - logvarsize)) + (gb_next_rand() >> (logvarsize - 1))
```

22. Variable x_v in this documentation means the variable whose information record is *varhead*[v]. But the user's variable '`x5`' might not be represented by *varhead*[5], because the ordering of variables can change as a program runs. If `x5` is really the variable in *varhead*[13], say, we will have *varmap*[5] = 13 and *varhead*[13].*name* = 5.

#define *topofvars* &*varhead*[*totvars*]

⟨ Global variables 5 ⟩ +≡
  var *varhead*[*varsize*]; /* basic info about each variable */
  var **tvar* = &*varhead*[*varsize*]; /* threshold for verbose printouts */
  int *varmap*[*varsize*]; /* the variable that has a given name */
  int *totvars*; /* the number of variables created */

23. Before any variables are used, we call the *createvars* routine to initialize the ones that the user asks for.

⟨ Subroutines 8 ⟩ +≡
  void *createvars*(**int** v)
  {
    register node $*p$, $*q$, $*r$;
    register var $*hv$ = &*varhead*[v];
    register int j, k;
    if ($\neg$*totvars*) ⟨ Create all the variables $(x_0, \ldots, x_v)$ 24 ⟩;
  }

24. We need a node at each level that means “tautology from here on,” i.e., all further branches lead to *topsink*. These nodes are called t_0, t_1, …, in printouts. Only t_0, which represents the constant 1, is considered external, reference-count-wise.

```
#define tautology node_(varhead[0].taut)   /* the constant function 1 */
⟨Create all the variables (x_0, ..., x_v) 24⟩ ≡
  {
    if (v + 1 ≥ varsize) {
      printf("Sorry, x%d is as high as I can go!\n", varsize − 2);
      exit(−4);
    }
    totvars = v + 1;
    o, oooo, botsink→index = (totvars ≪ (32 − logvarsize)) + (gb_next_rand() ≫ (logvarsize − 1));
      /* botsink has highest index */
    o, oooo, topsink→index = (totvars ≪ (32 − logvarsize)) + (gb_next_rand() ≫ (logvarsize − 1));
      /* so does topsink */
    for (k = 0; k ≤ v; k++) {
      o, varhead[k].base[0] = addr_(reserve_page());
        /* it won't be Λ, because leasesonlife = 1 before the call */
      ⟨Create a unique table for variable x_k with size 2 29⟩;
    }
    o, (topofvars)→taut = addr_(topsink);
    for (p = topsink, k = v; k ≥ 0; p = r, k−−) {
      r = unique_find(&varhead[k], p, p);
      oo, p→xref += 2;
      varhead[k].taut = addr_(r);   /* it won't be Λ either */
      p = unique_find(&varhead[k], botsink, topsink);
      oooo, botsink→xref++, topsink→xref++;
      o, varhead[k].elt = addr_(p);
      if (verbose & 2) printf(" %x=t%d, %x=e%d\n", id(r), k, id(p), k);
      if (k ≠ 0) oo, r→xref−−;
      oo, varhead[k].name = k, varmap[k] = k;
    }
    leasesonlife = 10;
  }
```

This code is used in section 23.

25. The simplest nonconstant Boolean expression is a projection function, x_v. Paradoxically, however, the ZDD for this expression is *not* so simple, because ZDDs are optimized for a different criterion of simplicity. We access it with the following subroutine, creating it from scratch if necessary. (Many applications of ZDDs don't need to mention the projection functions, because element functions and/or special-purpose routines are often good enough for building up the desired ZDD base.)

⟨ Subroutines 8 ⟩ +≡
```
node *projection(int v)
{
  register node *p, *q, *r;
  register var *hv = &varhead[v];
  register int j, k;
  if (¬hv→proj) {
    hv→proj = addr_(symfunc(node_(hv→elt), varhead, 1));
    if (verbose & 2) printf("␣%x=x%d\n", id(hv→proj), v);
  }
  return o, node_(hv→proj);
}
```

26. I sometimes like to use subroutines before I'm in the mood to write their innards. In such cases, pre-specifications like the ones given here allow me to procrastinate.

⟨ Templates for subroutines 26 ⟩ ≡
```
node *unique_find(var *v, node *l, node *h);
node *symfunc(node *p, var *v, int k);
```

See also sections 28 and 106.

This code is used in section 3.

27. Now, however, I'm ready to tackle the *unique_find* subroutine, which is one of the most crucial in the entire program. Given a variable v, together with node pointers l and h, we often want to see if the ZDD base contains a node (v,l,h)—namely, a branch on x_v with LO pointer l and HI pointer h. If no such node exists, we want to create it. The subroutine should return a pointer to that (unique) node. Furthermore, the reference counts of l and h should be decreased afterwards.

To do this task, we look for (l,h) in the unique table for v, using the hash code

$$(l\to index \ll 3) \oplus (h\to index \ll 2).$$

(This hash code is a multiple of 4, the size of each entry in the unique table.)

Several technicalities should be noted. First, no branch is needed when $h = botsink$. (This is the crucial difference between ZDDs and BDDs.) Second, we consider that a new reference is being made to the node returned, as well as to nodes l and h if a new node is created; the *xref* fields (reference counts) must be adjusted accordingly. Third, we might discover that the node exists, but it is dead; in other words, all prior links to it might have gone away, but we haven't discarded it yet. In such a case we should bring it back to life. Fourth, l and h will not become dead when their reference counts decrease, because the calling routine knows them. And finally, in the worst case we won't have room for a new node, so we'll have to return Λ. The calling routine must be prepared to cope with such failures (which we hope are only temporary).

The following inscrutable macros try to make my homegrown dynamic array addressing palatable. I have to admit that I didn't get them right the first time. Or even the second time. Or even . . . .

```
#define hashcode(l, h)  ((addr *)(size_t)(oo, ((l)→index ≪ 3) ⊕ ((h)→index ≪ 2)))
#define hashedcode(p)  hashcode(node_(p→lo), node_(p→hi))
#define addr__(x)  (*((addr *)(size_t)(x)))
#define fetchnode(v, k)  node_(addr__(v→base[(k) ≫ logpagesize] + ((k) & pagemask)))
#define storenode(v, k, p)  o, addr__(v→base[(k) ≫ logpagesize] + ((k) & pagemask)) = addr_(p)
```

⟨ Subroutines 8 ⟩ +≡

```
  node *unique_find(var *v, node *l, node *h)
  {
    register int j, k, mask, free;
    register addr *hash;
    register node *p, *r;
    if (h ≡ botsink) {      /* easy case */
      return oo, h→xref −−, l;     /* h→xref will still be ≥ 0 */
    }
  restart: o, mask = v→mask, free = v→free;
    for (hash = hashcode(l, h); ; hash++) {      /* ye olde linear probing */
      k = addr_(hash) & mask;
      oo, p = fetchnode(v, k);
      if (¬p) goto newnode;
      if (node_(p→lo) ≡ l ∧ node_(p→hi) ≡ h) break;
    }
    if (o, p→xref < 0) {
      deadnodes −−, o, p→xref = 0;     /* a lucky hit; its children are alive */
      return p;
    }
    oooo, l→xref −−, h→xref −−;
    return o, p→xref ++, p;
  newnode: ⟨ Periodically try to conserve space 30 ⟩;
    ⟨ Create a new node and return it 32 ⟩;
  }
```

28. ⟨ Templates for subroutines 26 ⟩ +≡
void *recursively_revive*(**node** *p); /* recursive resuscitation */
void *recursively_kill*(**node** *p); /* recursive euthanization */
void *collect_garbage*(**int** *level*); /* invocation of the recycler */

29. Before we can call *unique_find*, we need a hash table to work with. We start small.

#define *storenulls*(k) *(**long long** *)(**size_t**)(k) = 0_{LL};

⟨ Create a unique table for variable x_k with size 2 29 ⟩ ≡
o, *varhead*[k].*free* = 2, *varhead*[k].*mask* = 7;
storenulls(*varhead*[k].*base*[0]); /* both slots start out Λ */
zmems++;

This code is used in section 24.

30. A little timer starts ticking at the beginning of this program, and it advances whenever we reach the present point. Whenever the timer reaches a multiple of *timerinterval*, we pause to examine the memory situation, in an attempt to keep node growth under control.

Memory can be conserved in two ways. First, we can recycle all the dead nodes. That's a somewhat expensive proposition; but it's worthwhile if the number of such nodes is more than, say, 1/8 of the total number of nodes allocated. Second, we can try to change the ordering of the variables. The present program includes Rudell's "sifting algorithm" for dynamically improving the variable order; but it invokes that algorithm only under user control. Perhaps I will have time someday to make reordering more automatic.

#define *timerinterval* 1024
#define *deadfraction* 8

⟨ Periodically try to conserve space 30 ⟩ ≡
if ((++*timer* % *timerinterval*) ≡ 0) {
  if (*deadnodes* > *totalnodes*/*deadfraction*) {
    collect_garbage(0);
    goto *restart*; /* the hash table might now be different */
  }
}

This code is used in section 27.

31. ⟨ Global variables 5 ⟩ +≡
unsigned long long *timer*;

32. Brand-new nodes enter the fray here.

⟨ Create a new node and return it 32 ⟩ ≡
p = *reserve_node*( );
if (¬p) **goto** *cramped*; /* sorry, there ain't no more room */
if (−−*free* ≤ *mask* ≫ 4) {
  free_node(p);
  ⟨ Double the table size and **goto** *restart* 33 ⟩;
}
storenode(v, k, p); *o*, v→*free* = *free*;
initnewnode(p, v − *varhead*, l, h);
return p;
cramped: /* after failure, we need to keep the xrefs tidy */
deref(l); /* decrease l→*xref*, and recurse if it becomes dead */
deref(h); /* ditto for h */
return Λ;

This code is used in section 27.

33. We get to this part of the code when the table has become too dense. The density will now decrease from 3/4 to 3/8.

⟨ Double the table size and **goto** *restart* 33 ⟩ ≡

```
{
  register int newmask = mask + mask + 1, kk = newmask ≫ logpagesize;
  if (verbose & 256) printf("doubling␣the␣hash␣table␣for␣level␣%d(x%d)␣(%d␣slots)\n",
        v − varhead, v→name, (newmask + 1)/sizeof(addr));
  if (kk) ⟨ Reserve new all-Λ pages for the bigger table 34 ⟩
  else {
    for (k = v→base[0] + mask + 1; k < v→base[0] + newmask; k += sizeof(long long)) storenulls(k);
    zmems += (newmask − mask)/sizeof(long long);
  }
  ⟨ Rehash everything in the low half 35 ⟩;
  v→mask = newmask;    /* mems are counted after restarting */
  v→free = free + 1 + (newmask − mask)/sizeof(addr);
  goto restart;
}
```

This code is used in sections 32, 136, and 140.

34. `#define maxmask ((1 ≪ logmaxhashsize) ∗ sizeof(addr) − 1)    /* the biggest possible mask */`

⟨ Reserve new all-Λ pages for the bigger table 34 ⟩ ≡

```
{
  if (newmask > maxmask) {    /* too big: can't go there */
    if (verbose & (2 + 256 + 512))
      printf("profile␣limit␣reached␣for␣level␣%d(x%d)\n", v − varhead, v→name);
    goto cramped;
  }
  for (k = (mask ≫ logpagesize) + 1; k ≤ kk; k++) {
    o, v→base[k] = addr_(reserve_page());
    if (¬v→base[k]) {    /* oops, we're out of space */
      for (k−−; k > mask ≫ logpagesize; k−−) {
        o, free_page(page_(v→base[k]));
      }
      goto cramped;
    }
    for (j = v→base[k]; j < v→base[k] + pagesize; j += sizeof(long long)) storenulls(j);
    zmems += pagesize/sizeof(long long);
  }
}
```

This code is used in section 33.

35. Some subtle cases can arise at this point. For example, consider the hash table (a, Λ, Λ, b), with hash$(a) = 3$ and hash$(b) = 7$; when doubling the size, we need to rehash a twice, going from the doubled-up table $(a, \Lambda, \Lambda, b, \Lambda, \Lambda, \Lambda, \Lambda)$ to $(\Lambda, \Lambda, \Lambda, b, a, \Lambda, \Lambda, \Lambda)$ to $(\Lambda, \Lambda, \Lambda, \Lambda, a, \Lambda, \Lambda, b)$ to $(\Lambda, \Lambda, \Lambda, a, \Lambda, \Lambda, \Lambda, b)$.

I learned this interesting algorithm from Rick Rudell.

```
⟨ Rehash everything in the low half 35 ⟩ ≡
  for (k = 0; k < newmask; k += sizeof(addr)) {
    oo, r = fetchnode(v, k);
    if (r) {
      storenode(v, k, Λ);    /* prevent propagation past this slot */
      for (o, hash = hashedcode(r); ; hash++) {
        j = addr_(hash) & newmask;
        oo, p = fetchnode(v, j);
        if (¬p) break;
      }
      storenode(v, j, r);
    } else if (k > mask) break;    /* see the example above */
  }
```

This code is used in section 33.

36. While I've got linear probing firmly in mind, I might as well write a subroutine that will be needed later for garbage collection. The *table_purge* routine deletes all dead nodes that branch on a given variable x_v.

```
⟨ Subroutines 8 ⟩ +≡
  void table_purge(var *v)
  {
    register int free, i, j, jj, k, kk, mask, newmask, oldtotal;
    register node *p, *r;
    register addr *hash;
    o, mask = v→mask, free = v→free;
    oldtotal = totalnodes;
    for (k = 0; k < mask; k += sizeof(addr)) {
      oo, p = fetchnode(v, k);
      if (p ∧ p→xref < 0) {
        free_node(p);
        ⟨ Remove entry k from the hash table 37 ⟩;
      }
    }
    deadnodes −= oldtotal − totalnodes, free += oldtotal − totalnodes;
    ⟨ Downsize the table if only a few entries are left 38 ⟩;
    o, v→free = free;
  }
```

37. Deletion from a linearly probed hash table is tricky, as noted in Algorithm 6.4R of TAOCP. Here I can speed that algorithm up slightly, because there's no need to move dead entries that will be deleted later.

Furthermore, if I do meet a dead entry, I can take a slightly tricky shortcut and continue the removals.

⟨ Remove entry k from the hash table 37 ⟩ ≡

```
do {
  for (kk = k, j = k + sizeof(addr), k = 0; ; j += sizeof(addr)) {
    jj = j & mask;
    oo, p = fetchnode(v, jj);
    if (¬p) break;
    if (p→xref ≥ 0) {
      o, i = addr_(hashedcode(p)) & mask;
      if ((i ≤ kk) + (jj < i) + (kk < jj) > 1) storenode(v, kk, p), kk = jj;
    } else if (¬k) k = j, free_node(p);      /* shortcut */
  }
  storenode(v, kk, Λ);
} while (k);
k = j;     /* the last run through that loop saw no dead nodes */
```

This code is used in section 36.

38. At least one node, $v\to elt$, branches on x_v at this point.

⟨ Downsize the table if only a few entries are left 38 ⟩ ≡

```
k = (mask ≫ 2) + 1 − free;     /* this many nodes still branch on x_v */
for (newmask = mask; (newmask ≫ 5) ≥ k; newmask ≫= 1) ;
if (newmask ≠ mask) {
  if (verbose & 256) printf("downsizing␣the␣hash␣table␣for␣level␣%d(x%d)␣(%d␣slots)\n",
        v − varhead, v→name, (newmask + 1)/sizeof(addr));
  free −= (mask − newmask) ≫ 2;
  ⟨ Rehash everything in the upper half 39 ⟩;
  for (k = mask ≫ logpagesize; k > newmask ≫ logpagesize; k−−) o, free_page(page_(v→base[k]));
  v→mask = newmask;
}
```

This code is used in section 36.

39. Finally, another algorithm learned from Rudell. To prove its correctness, one can verify the following fact: Any entries that wrapped around from the upper half to the bottom in the original table will still wrap around in the smaller table.

⟨ Rehash everything in the upper half 39 ⟩ ≡

```
for (k = newmask + 1; k < mask; k += sizeof(addr)) {
  oo, r = fetchnode(v, k);
  if (r) {
    for (o, hash = hashedcode(r); ; hash++) {
      j = addr_(hash) & newmask;
      oo, p = fetchnode(v, j);
      if (¬p) break;
    }
    storenode(v, j, r);
  }
}
```

This code is used in section 38.

40. The cache. The other principal data structure we need, besides the ZDD base itself, is a software cache that helps us avoid repeating the calculations that we've already done. If, for example, f and g are nodes of the ZDD for which we've already computed $h = f \wedge g$, the cache should contain the information that $f \wedge g$ is known to be node h.

But that description is only approximately correct, because the cost of forgetting the value of $f \wedge g$ is less than the cost of building a fancy data structure that is able to remember every result. (If we forget only a few things, we need to do only a few recomputations.) Therefore we adopt a simple scheme that is designed to be reliable most of the time, yet not perfect: We look for $f \wedge g$ in only one position within the cache, based on a hash code. If two or more results happen to hash to the same cache slot, we remember only the most recent one.

Every entry of the cache consists of four tetrabytes, called f, g, h, and r. The last of these, r, is nonzero if and only if the cache entry is meaningful; in that case r points to a ZDD node, the result of an operation encoded by f, g, and h. This (f, g, h) encoding has several variants:

- If $0 \le h \le maxbinop$, then h denotes a binary operation on the ZDD nodes f and g. For example, $h = 1$ stands for $\wedge$. The binary operations currently implemented are: disproduct (0), and (1), but-not (2), product (5), xor (6), or (7), coproduct (8), quotient (9), remainder (10), delta (11).
- Otherwise (f, g, h) encodes a ternary operation on the three ZDD nodes f, g, $h \mathbin{\&} -16$. The four least-significant bits of h are used to identify the ternary operation involved: if-then-else (0), median (1), and-and (2), zdd-build (3), symfunc (4), not-yet-implemented (5–15).

#define *memo_*(*a*) ((**memo** ∗)(**size_t**)(*a*))

⟨ Type definitions 11 ⟩ +≡
```
typedef struct memo_struct {
  addr f;   /* first operand */
  addr g;   /* second operand */
  addr h;   /* third operand and/or operation code */
  addr r;   /* result */
} memo;
```

41. The cache always occupies 2^e pages of the dynamic memory, for some integer $e \ge 0$. If we have leisure to choose this size, we pick the smallest $e \ge 0$ such that the cache has at least $\max(4m, n/4)$ slots, where m is the number of nonempty items in the cache and n is the number of live nodes in the ZDD. Furthermore, the cache size will double whenever the number of cache insertions reaches a given threshold.

```
#define logmaxcachepages 15   /* shouldn't be large if logvarsize is large */
#define maxcachepages (1 << logmaxcachepages)
#define cacheslotsperpage (pagesize/sizeof(memo))
#define maxbinop 15
```

⟨ Global variables 5 ⟩ +≡
```
addr cachepage[maxcachepages];   /* base addresses for the cache */
int cachepages;   /* the current number of pages in the cache */
int cacheinserts;   /* the number of times we've inserted a memo */
int threshold;   /* the number of inserts that trigger cache doubling */
int cachemask;   /* index of the first slot following the cache, minus 1 */
```

42. The following subroutines, useful for debugging, print out the cache contents in symbolic form. If p points to a node, *id*(p) is $p - botsink$.

```
#define id(a) (((size_t)(a) − (size_t) mem)/sizeof(node))   /* node number in mem */
⟨Subroutines 8⟩ +≡
  void print_memo(memo *m)
  {
    printf("%x", id(m→f));
    if (m→h ≤ maxbinop) printf("%s%x", binopname[m→h], id(m→g));
    else printf("%s%x%s%x", ternopname1[m→h & #f], id(m→g), ternopname2[m→h & #f], id(m→h));
    printf("=%x\n", id(m→r));
  }
  void print_cache(void)
  {
    register int k;
    register memo *m;
    for (k = 0; k < cachepages; k++)
      for (m = memo_(cachepage[k]); m < memo_(cachepage[k]) + cacheslotsperpage; m++)
        if (m→r) print_memo(m);
  }
```

43. Many of the symbolic names here are presently unused. I've filled them in just to facilitate extensions to this program.

```
⟨Global variables 5⟩ +≡
  char *binopname[] = {"+","&",">","!","<","*","^","|","\"","/","%","_",":","$",";",","};
  char *ternopname1[] = {"?",".","&","!","@","#","$","%","*","<","-","+","|","/","\\","~"};
  char *ternopname2[] = {":",".","&",":","@","#","$","%","*","<","-","+","|","/","\\","~"};
```

44. The threshold is set to half the total number of cache slots, because this many random insertions will keep about $e^{-1/2} \approx 61\%$ of the cache slots unclobbered. (If p denotes this probability, a random large binary tree will need about E steps to recalculate a lost result, where $E = p \cdot 1 + (1-p) \cdot (1+2E)$; hence we want $p > 1/2$ to avoid blowup, and $E = 1/(2p-1)$.)

⟨ Subroutines 8 ⟩ +≡
```
int choose_cache_size(int items)
{
  register int k, slots;
  k = 1, slots = cacheslotsperpage;
  while (4 * slots < totalnodes − deadnodes ∧ k < maxcachepages) k <<= 1, slots <<= 1;
  while (slots < 4 * items ∧ k < maxcachepages) k <<= 1, slots <<= 1;
  return k;
}
void cache_init(void)
{
  register int k;
  register memo *m;
  cachepages = choose_cache_size(0);
  if (verbose & (8 + 16 + 32 + 512))
    printf("initializing␣the␣cache␣(%d␣page%s)\n", cachepages, cachepages ≡ 1 ? "" : "s");
  for (k = 0; k < cachepages; k++) {
    o, cachepage[k] = addr_(reserve_page());
    if (¬cachepage[k]) {
      fprintf(stderr, "(trouble␣allocating␣cache␣pages!)\n");
      for (k−−; (k + 1) & k; k−−) o, free_page(page_(cachepage[k]));
      cachepages = k + 1;
      break;
    }
    for (m = memo_(cachepage[k]); m < memo_(cachepage[k]) + cacheslotsperpage; m++) m→r = 0;
    zmems += cacheslotsperpage;
  }
  cachemask = (cachepages ≪ logpagesize) − 1;
  cacheinserts = 0;
  threshold = 1 + (cachepages * cacheslotsperpage)/2;
}
```

45. ⟨ Initialize everything 6 ⟩ +≡
```
cache_init();
```

46. Here's how we look for a memo in the cache. Memos might point to dead nodes, as long as those nodes still exist.

A simple hash function is adequate for caching, because no clustering can occur.

No mems are charged for computing *cachehash*, because we assume that the calling routine has taken responsibility for accessing $f\to index$ and $g\to index$.

#define $cachehash(f,g,h)$ $((f)\to index \ll 4) \oplus (((h) \mathrel{?} (g)\to index : addr\_(g)) \ll 5) \oplus (addr\_(h) \ll 6)$
#define $thememo(s)$ $memo\_(cachepage[((s) \mathbin{\&} cachemask) \gg logpagesize] + ((s) \mathbin{\&} pagemask))$

⟨ Subroutines 8 ⟩ +≡

```
node *cache_lookup(node *f, node *g, node *h)
{
  register node *r;
  register memo *m;
  register addr slot = cachehash(f, g, h);
  o, m = thememo(slot);
  o, r = node_(m→r);
  if (¬r) return Λ;
  if (o, node_(m→f) ≡ f ∧ node_(m→g) ≡ g ∧ node_(m→h) ≡ h) {
    if (verbose & 8) {
      printf("hit␣%x:␣", (slot & cachemask)/sizeof(memo));
      print_memo(m);
    }
    if (o, r→xref < 0) {
      recursively_revive(r);
      return r;
    }
    return o, r→xref++, r;
  }
  return Λ;
}
```

47. Insertion into the cache is even easier, except that we might want to double the cache size while we're at it.

⟨ Subroutines 8 ⟩ +≡

```
void cache_insert(node *f, node *g, node *h, node *r)
{
  register memo *m, *mm;
  register int k;
  register int slot = cachehash(f, g, h);
  if (h) oo; else o;      /* mems for computing cachehash */
  if (++cacheinserts ≥ threshold) ⟨ Double the cache size 48 ⟩;
  o, m = thememo(slot);
  if ((verbose & 16) ∧ m→r) {
    printf("lose␣%x:␣", (slot & cachemask)/sizeof(memo));
    print_memo(m);
  }
  oo, m→f = addr_(f), m→g = addr_(g), m→h = addr_(h), m→r = addr_(r);
  if (verbose & 32) {
    printf("set␣%x:␣", (slot & cachemask)/sizeof(memo));
    print_memo(m);
  }
}
```

48. ⟨ Double the cache size 48 ⟩ ≡

```
if (cachepages < maxcachepages) {
  if (verbose & (8 + 16 + 32 + 512)) printf("doubling␣the␣cache␣(%d␣pages)\n", cachepages ≪ 1);
  for (k = cachepages; k < cachepages + cachepages; k++) {
    o, cachepage[k] = addr_(reserve_page());
    if (¬cachepage[k]) {      /* sorry, we can't double the cache after all */
      fprintf(stderr, "(trouble␣doubling␣cache␣pages!)\n");
      for (k−−; k ≥ cachepages; k−−) o, free_page(page_(cachepage[k]));
      goto done;
    }
    for (m = memo_(cachepage[k]); m < memo_(cachepage[k]) + cacheslotsperpage; m++) m→r = 0;
    zmems += cacheslotsperpage;
  }
  cachepages ≪= 1;
  cachemask += cachemask + 1;
  threshold = 1 + (cachepages * cacheslotsperpage)/2;
  ⟨ Recache the items in the bottom half 49 ⟩;
}
done:
```

This code is used in section 47.

49. ⟨ Recache the items in the bottom half 49 ⟩ ≡

```
for (k = cachepages ≫ 1; k < cachepages; k++) {
  for (o, m = memo_(cachepage[k]); m < memo_(cachepage[k]) + cacheslotsperpage; m++)
    if (o, m→r) {
      if (m→h) oo; else o;      /* mems for computing cachehash */
      oo, mm = thememo(cachehash(node_(m→f), node_(m→g), node_(m→h)));
      if (m ≠ mm) {
        oo, *mm = *m;
        o, m→r = 0;
      }
    }
}
```

This code is used in section 48.

50. Before we purge elements from the unique tables, we need to purge all references to dead nodes from the cache.

⟨ Subroutines 8 ⟩ +≡
```
void cache_purge(void)
{
  register int k, items, newcachepages;
  register memo *m, *mm;
  for (k = items = 0; k < cachepages; k++) {
    for (m = memo_(cachepage[k]); m < memo_(cachepage[k]) + cacheslotsperpage; m++)
      if (o, m→r) {
        if ((o, node_(m→r)→xref < 0) ∨ (oo, node_(m→f)→xref < 0)) goto purge;
        if (o, node_(m→g)→xref < 0) goto purge;
        if (m→h > maxbinop ∧ (o, node_(m→h & −#10)→xref < 0)) goto purge;
        items++; continue;
      purge: o, m→r = 0;
      }
  }
  if (verbose & (8 + 16 + 32 + 512)) printf("purging␣the␣cache␣(%d␣items␣left)\n", items);
  ⟨ Downsize the cache if it has now become too sparse 51 ⟩;
  cacheinserts = items;
}
```

51. ⟨ Downsize the cache if it has now become too sparse 51 ⟩ ≡
```
newcachepages = choose_cache_size(items);
if (newcachepages < cachepages) {
  if (verbose & (8 + 16 + 32 + 512))
    printf("downsizing␣the␣cache␣(%d␣page%s)\n", newcachepages, newcachepages ≡ 1 ? "" : "s");
  cachemask = (newcachepages ≪ logpagesize) − 1;
  for (k = newcachepages; k < cachepages; k++) {
    for (o, m = memo_(cachepage[k]); m < memo_(cachepage[k]) + cacheslotsperpage; m++)
      if (o, m→r) {
        if (m→h) oo; else o;   /* mems for computing cachehash */
        oo, mm = thememo(cachehash(node_(m→f), node_(m→g), node_(m→h)));
        if (m ≠ mm) {
          oo, *mm = *m;
        }
      }
    free_page(page_(cachepage[k]));
  }
  cachepages = newcachepages;
  threshold = 1 + (cachepages * cacheslotsperpage)/2;
}
```
This code is used in section 50.

52. ZDD structure. The reader of this program ought to be familiar with the basics of ZDDs, namely the facts that a ZDD base consists of two sink nodes together with an unlimited number of branch nodes, where each branch node (v,l,h) names a variable x_v and points to other nodes l and h that correspond to the cases where $x_v = 0$ and $x_v = 1$. The variables on every path have increasing rank v, and no two nodes have the same (v,l,h). Furthermore, $h \neq \mathit{botsink}$.

Besides the nodes of the ZDD, this program deals with external pointers f_j for $0 \le j < \mathit{extsize}$. Each f_j is either Λ or points to a ZDD node.

```
#define extsize 10000
⟨ Global variables 5 ⟩ +≡
  node *f[extsize];   /* external pointers to functions in the ZDD base */
```

53. Sometimes we want to mark the nodes of a subfunction temporarily. The following routine sets the leading bit of the *xref* field in all nodes reachable from p.

```
⟨ Subroutines 8 ⟩ +≡
  void mark(node *p)
  {
    rmems++;   /* track recursion overhead */
  restart: if (o, p→xref ≥ 0) {
      o, p→xref ⊕= #80000000;
      ooo, mark(node_(p→lo));   /* two extra mems to save and restore p */
      o, p = node_(p→hi);
      goto restart;   /* tail recursion */
    }
  }
```

54. We need to remove those marks soon after *mark* has been called, because the *xref* field is really supposed to count references.

```
⟨ Subroutines 8 ⟩ +≡
  void unmark(node *p)
  {
    rmems++;   /* track recursion overhead */
  restart: if (o, p→xref < 0) {
      o, p→xref ⊕= #80000000;
      ooo, unmark(node_(p→lo));   /* two extra mems to save and restore p */
      o, p = node_(p→hi);
      goto restart;   /* tail recursion */
    }
  }
```

55. Here's a simple routine that prints out the current ZDDs, in order of the variables in branch nodes. If the *marked* parameter is nonzero, the output is restricted to branch nodes whose *xref* field is marked. Otherwise all nodes are shown, with nonzero *xref*s in parentheses.

```
#define thevar(p) (&varhead[varpart((p)→index)])
#define print_node(p) printf("%x:␣(~%d?%x:%x)", id(p), thevar(p)→name, id((p)→lo), id((p)→hi))
⟨ Subroutines 8 ⟩ +≡
  void print_base(int marked)
  {
    register int j, k;
    register node *p;
    register var *v;

    for (v = varhead; v < topofvars; v++) {
      for (k = 0; k < v→mask; k += sizeof(addr)) {
        p = fetchnode(v, k);
        if (p ∧ (¬marked ∨ (p→xref + 1) < 0)) {
          print_node(p);
          if (marked ∨ p→xref ≡ 0) printf("\n");
          else printf("␣(%d)\n", p→xref);
        }
      }
      if (¬marked) {
        printf("t%d=%x\ne%d=%x\n", v→name, id(v→taut), v→name, id(v→elt));
        if (v→proj) printf("x%d=%x\n", v→name, id(v→proj));
      }
    }
    if (¬marked) {    /* we also print the external functions */
      for (j = 0; j < extsize; j++)
        if (f[j]) printf("f%d=%x\n", j, id(f[j]));
    }
  }
```

56. The masking feature is useful when we want to print out only a single ZDD.

```
⟨ Subroutines 8 ⟩ +≡
  void print_function(node *p)
  {
    unsigned long long savemems = mems, savermems = rmems;
      /* mems aren't counted while printing */
    if (p ≡ botsink ∨ p ≡ topsink) printf("%d\n", p − botsink);
    else if (p) {
      mark(p);
      print_base(1);
      unmark(p);
    }
    mems = savemems, rmems = savermems;
  }
```

57. ⟨ Subroutines 8 ⟩ +≡

```
void print_profile(node *p)
{
  unsigned long long savemems = mems, savermems = rmems;
  register int j, k, tot, bot = 0;
  register var *v;
  if (¬p) printf(" 0\n");    /* vacuous */
  else if (p ≤ topsink) printf(" 1\n");    /* constant */
  else {
    tot = 0;
    mark(p);
    for (v = varhead; v < topofvars; v++) {
      ⟨ Print the number of marked nodes that branch on v 58 ⟩;
    }
    unmark(p);
    printf(" %d (total %d)\n", bot + 1, tot + bot + 1);    /* the sinks */
  }
  mems = savemems, rmems = savermems;
}
```

58. ⟨ Print the number of marked nodes that branch on v 58 ⟩ ≡

```
for (j = k = 0; k < v→mask; k += sizeof(addr)) {
  register node *q = fetchnode(v, k);
  if (q ∧ (q→xref + 1) < 0) {
    j++;
    if (node_(q→lo) ≡ botsink) bot = 1;
  }
}
printf(" %d", j);
tot += j;
```

This code is used in section 57.

59. In order to deal efficiently with large ZDDs, we've introduced highly redundant data structures, including things like hash tables and the cache. Furthermore, we assume that every ZDD node p has a redundant field $p\to xref$, which counts the total number of branch nodes, external nodes, and projection functions that point to p, minus 1.

Bugs in this program might easily corrupt the data structure by putting it into an inconsistent state. Yet the inconsistency might not show up at the time of the error; the computer might go on to execute millions of instructions before the anomalies lead to disaster.

Therefore I've written a *sanity_check* routine, which laboriously checks the integrity of all the data structures. This routine should help me to pinpoint problems readily whenever I make mistakes. And besides, the *sanity_check* calculations document the structures in a way that should be especially helpful when I reread this program a year from now.

Even today, I think that the very experience of writing *sanity_check* has made me much more familiar with the structures themselves. This reinforced knowledge will surely be valuable as I write the rest of the code.

```
#define includesanity 1

⟨Subroutines 8⟩ +≡
#if includesanity
  unsigned int sanitycount;    /* how many sanity checks have been started? */
  void sanity_check(void)
  {
    register node *p, *q;
    register int j, k, count, extra;
    register var *v;
    unsigned long long savemems = mems;

    sanitycount++;
    ⟨Build the shadow memory 61⟩;
    ⟨Check the reference counts 67⟩;
    ⟨Check the unique tables 69⟩;
    ⟨Check the cache 70⟩;
    ⟨Check the list of free pages 71⟩;
    mems = savemems;
  }
#endif
```

60. Sanity checking is done with a "shadow memory" *smem*, which is just as big as *mem*. If p points to a node in *mem*, there's a corresponding "ghost" in *smem*, pointed to by $q = ghost(p)$. The ghost nodes have four fields *lo*, *hi*, *xref*, and *index*, just like ordinary nodes do; but the meanings of those fields are quite different: $q\to xref$ is -1 if node p is in the free list, otherwise $q\to xref$ is a backpointer to a field that points to p. If $p\to lo$ points to r, then $q\to lo$ will be a backpointer that continues the list of pointers to r that began with the *xref* field in r's ghost; and there's a similar relationship between $p\to hi$ and $q\to hi$. (Thus we can find all nodes that point to p.) Finally, $q\to index$ counts the number of references to p from external pointers and projection functions.

```
#define ghost(p) node_((size_t)(p) − (size_t) mem + (size_t) smem)

⟨Global variables 5⟩ +≡
#if includesanity
  char smem[memsize];    /* the shadow memory */
#endif
```

61. **#define** $complain(complaint)$
  { $printf$("!␣%s␣in␣node␣", $complaint$); $print\_node(p)$; $printf$("\n"); }
#define $legit(p)$
  $(((\mathbf{size\_t})(p)\ \&\ (\mathbf{sizeof}(\mathbf{node})-1)) \equiv 0 \land (p) < nodeptr \land (p) \geq botsink \land ghost(p)\to xref \neq -1)$
#define $superlegit(p)$
  $(((\mathbf{size\_t})(p)\ \&\ (\mathbf{sizeof}(\mathbf{node})-1)) \equiv 0 \land (p) < nodeptr \land (p) > topsink \land ghost(p)\to xref \neq -1)$

⟨ Build the shadow memory 61 ⟩ ≡

```
for (p = botsink; p < nodeptr; p++) ghost(p)→xref = 0, ghost(p)→index = −1;
⟨ Check the list of free nodes 65 ⟩;
⟨ Compute the ghost index fields 66 ⟩;
for (count = 2, p = topsink + 1; p < nodeptr; p++)
  if (ghost(p)→xref ≠ −1) {
    count++;
    if (¬legit(node_(p→lo)) ∨ ¬legit(node_(p→hi))) complain("bad␣pointer")
    else if (node_(thevar(p)→elt) ≡ Λ) complain("bad␣var")
    else if (node_(p→hi) ≡ botsink) complain("hi=bot")
    else {
      ⟨ Check that p is findable in the unique table 64 ⟩;
      if (node_(p→lo) > topsink ∧ thevar(p) ≥ thevar(node_(p→lo))) complain("bad␣lo␣rank");
      if (node_(p→hi) > topsink ∧ thevar(p) ≥ thevar(node_(p→hi))) complain("bad␣hi␣rank");
      if (p→xref ≥ 0) {    /* dead nodes don't point */
        q = ghost(p);
        q→lo = ghost(p→lo)→xref, ghost(p→lo)→xref = addr_(&(p→lo));
        q→hi = ghost(p→hi)→xref, ghost(p→hi)→xref = addr_(&(p→hi));
      }
    }
  }
if (count ≠ totalnodes) printf("!␣totalnodes␣should␣be␣%d,␣not␣%d\n", count, totalnodes);
if (extra ≠ totalnodes) printf("!␣%d␣nodes␣have␣leaked\n", extra − totalnodes);
```

This code is used in section 59.

62. The macros above and the *who_points_to* routine below rely on the fact that **sizeof**(**node**) = 16.

63. ⟨ Initialize everything 6 ⟩ +≡

```
if (sizeof(node) ≠ 16) {
  fprintf(stderr, "Sorry,␣I␣assume␣that␣sizeof(node)␣is␣16!\n");
  exit(−3);
}
```

64. ⟨ Check that p is findable in the unique table 64 ⟩ ≡

```
{
  register addr *hash;
  register var *v = thevar(p);
  j = v→mask;
  for (hash = hashcode(node_(p→lo), node_(p→hi)); ; hash++) {
    k = addr_(hash) & j;
    q = fetchnode(v, k);
    if (¬q) break;
    if (q→lo ≡ p→lo ∧ q→hi ≡ p→hi) break;
  }
  if (q ≠ p) complain("unfindable␣(lo,hi)");
  addr__((size_t)(v→base[k ≫ logpagesize] + (k & pagemask)) − (size_t) mem + (size_t)
      smem) = sanitycount;
}
```

This code is used in section 61.

65. ⟨ Check the list of free nodes 65 ⟩ ≡

```
extra = nodeptr − botsink;
for (p = nodeavail; p; p = node_(p→xref)) {
  if (¬superlegit(p)) printf("!␣illegal␣node␣%x␣in␣the␣list␣of␣free␣nodes\n", id(p));
  else extra−−, ghost(p)→xref = −1;
}
```

This code is used in section 61.

66. ⟨ Compute the ghost index fields 66 ⟩ ≡

```
ghost(botsink)→index = ghost(topsink)→index = 0;
for (v = varhead; v < topofvars; v++) {
  if (v→proj) {
    if (¬superlegit(node_(v→proj)))
      printf("!␣illegal␣projection␣function␣for␣level␣%d\n", v − varhead);
    else ghost(v→proj)→index++;
  }
  if (¬superlegit(node_(v→taut)))
    printf("!␣illegal␣tautology␣function␣for␣level␣%d\n", v − varhead);
  if (¬superlegit(node_(v→elt)))
    printf("!␣illegal␣projection␣function␣for␣level␣%d\n", v − varhead);
  else ghost(v→elt)→index++;
}
if (totvars) ghost(varhead[0].taut)→index++;    /* tautology is considered external */
for (j = 0; j < extsize; j++)
  if (f[j]) {
    if (f[j] > topsink ∧ ¬superlegit(f[j])) printf("!␣illegal␣external␣pointer␣f%d\n", j);
    else ghost(f[j])→index++;
  }
```

This code is used in section 61.

67. ⟨ Check the reference counts 67 ⟩ ≡

```
for (p = botsink, count = 0; p < nodeptr; p++) {
  q = ghost(p);
  if (q→xref ≡ −1) continue;   /* p is free */
  for (k = q→index, q = node_(q→xref); q; q = node_(addr__(ghost(q)))) k++;
  if (p→xref ≠ k) printf("!␣%x->xref␣should␣be␣%d,␣not␣%d\n", id(p), k, p→xref);
  if (k < 0) count++;   /* p is dead */
}
if (count ≠ deadnodes) printf("!␣deadnodes␣should␣be␣%d,␣not␣%d\n", count, deadnodes);
```

This code is used in section 59.

68. If a reference count turns out to be wrong, I'll probably want to know why. The following subroutine provides additional clues.

⟨ Subroutines 8 ⟩ +≡

```
#if includesanity
  void who_points_to(node *p)
  {
    register addr q;   /* the address of a lo or hi field in a node */
    for (q = addr_(ghost(p)→xref); q; q = addr__(ghost(q))) {
      print_node(node_(q & −sizeof(node)));
      printf("\n");
    }
  }
#endif
```

69. We've seen that every superlegimate node is findable in the proper unique table. Conversely, we want to check that everything is those tables is superlegitimate, and found.

```
#define badpage(p) ((p) < pageptr ∨ (p) ≥ topofmem)
```

⟨ Check the unique tables 69 ⟩ ≡

```
extra = topofmem − pageptr;   /* this many pages allocated */
for (v = varhead; v < topofvars; v++) {
  for (k = 0; k ≤ v→mask ≫ logpagesize; k++)
    if (badpage(page_(v→base[k])))
      printf("!␣bad␣page␣base␣%x␣in␣unique␣table␣for␣level␣%d\n", id(v→base[k]), v − varhead);
  extra −= 1 + (v→mask ≫ logpagesize);
  for (k = count = 0; k < v→mask; k += sizeof(addr)) {
    p = fetchnode(v, k);
    if (¬p) count++;
    else {
      if (addr__((size_t)(v→base[k ≫ logpagesize] + (k & pagemask)) − (size_t) mem + (size_t)
            smem) ≠ sanitycount)
        printf("!␣extra␣node␣%x␣in␣unique␣table␣for␣level␣%d\n", id(p), v − varhead);
      if (¬superlegit(p))
        printf("!␣illegal␣node␣%x␣in␣unique␣table␣for␣level␣%d\n", id(p), v − varhead);
      else if (varpart(p→index) ≠ v − varhead) complain("wrong␣var");
    }
  }
  if (count ≠ v→free)
    printf("!␣unique␣table␣%d␣has␣%d␣free␣slots,␣not␣%d\n", v − varhead, count, v→free);
}
```

This code is used in section 59.

70. The fields in cache memos that refer to nodes should refer to legitimate nodes.

⟨ Check the cache 70 ⟩ ≡

```
{
  register memo *m;
  extra -= 1 + (cachemask >> logpagesize);
  for (k = 0; k < cachepages; k++) {
    if (badpage(page_(cachepage[k])))
      printf("! bad page base %x in the cache\n", id(cachepage[k]));
    for (m = memo_(cachepage[k]); m < memo_(cachepage[k]) + cacheslotsperpage; m++)
      if (m->r) {
        if (¬legit(node_(m->r))) goto nogood;
        if (¬legit(node_(m->f))) goto nogood;
        if (¬legit(node_(m->g))) goto nogood;
        if (m->h > maxbinop ∧ ¬legit(node_(m->h & -#10))) goto nogood;
      }
    continue;
  nogood: printf("! bad node in cache entry "); print_memo(m);
  }
}
```

This code is used in section 59.

71. Finally, *sanity_check* ensures that we haven't forgotten to free unused pages, nor have we freed a page that was already free.

⟨ Check the list of free pages 71 ⟩ ≡

```
{
  register page *p = pageavail;
  while (p ∧ extra > 0) {
    if (badpage(p)) printf("! bad free page %x\n", id(p));
    p = page_(p->dat[0]), extra--;
  }
  if (extra > 0) printf("! %d pages have leaked\n", extra);
  else if (p) printf("! the free pages form a loop\n");
}
```

This code is used in section 59.

72. The following routine brings a dead node back to life. It also increases the reference counts of the node's children, and resuscitates them if they were dead.

⟨ Subroutines 8 ⟩ +≡
```
void recursively_revive(node *p)
{
  register node *q;
  rmems++;   /* track recursion overhead */
restart: if (verbose & 4) printf("reviving␣%x\n", id(p));
  o, p→xref = 0;
  deadnodes−−;
  q = node_(p→lo);
  if (o, q→xref < 0) oooo, recursively_revive(q);
  else o, q→xref++;
  p = node_(p→hi);
  if (o, p→xref < 0) goto restart;   /* tail recursion */
  else o, p→xref++;
}
```

73. Conversely, we sometimes must go the other way, with as much dignity as we can muster.

```
#define deref(p)
        if (o, (p)→xref ≡ 0) recursively_kill(p); else o, (p)→xref−−
```

⟨ Subroutines 8 ⟩ +≡
```
void recursively_kill(node *p)
{
  register node *q;
  rmems++;   /* track recursion overhead */
restart: if (verbose & 4) printf("burying␣%x\n", id(p));
  o, p→xref = −1;
  deadnodes++;
  q = node_(p→lo);
  if (o, q→xref ≡ 0) oooo, recursively_kill(q);
  else o, q→xref−−;
  p = node_(p→hi);
  if (o, p→xref ≡ 0) goto restart;   /* tail recursion */
  else o, p→xref−−;
}
```

74. Binary operations. OK, now we've got a bunch of powerful routines for making and maintaining ZDDs, and it's time to have fun. Let's start with a typical synthesis routine, which constructs the ZDD for $f \land g$ from the ZDDs for f and g.

The general pattern is to have a top-level subroutine and a recursive subroutine. The top-level one updates overall status variables and invokes the recursive one; and it keeps trying, if temporary setbacks arise.

The recursive routine exits quickly if given a simple case. Otherwise it checks the cache, and calls itself if necessary. I write the recursive routine first, since it embodies the guts of the computation.

The top-level routines are rather boring, so I'll defer them till later.

Incidentally, I learned the C language long ago, and didn't know until recently that it's now legal to modify the formal parameters to a function. (Wow!)

```
⟨ Subroutines 8 ⟩ +≡
  node *and_rec(node *f, node *g)
  {
    var *v, *vf, *vg;
    node *r, *r0, *r1;
    oo, vf = thevar(f), vg = thevar(g);
    while (vf ≠ vg) {
      if (vf < vg) {
        if (g ≡ botsink) return oo, g→xref++, g;    /* f ∧ 0 = 0 */
        oo, f = node_(f→lo), vf = thevar(f);    /* wow */
      }
      else if (f ≡ botsink) return oo, f→xref++, f;    /* 0 ∧ g = 0 */
      else oo, g = node_(g→lo), vg = thevar(g);
    }
    if (f ≡ g) return oo, f→xref++, f;    /* f ∧ f = f */
    if (f > g) r = f, f = g, g = r;
    if (o, f ≡ node_(vf→taut)) return oo, g→xref++, g;    /* 1 ∧ g = g */
    if (g ≡ node_(vf→taut)) return oo, f→xref++, f;    /* f ∧ 1 = f */
    r = cache_lookup(f, g, node_(1));    /* we've already fetched f→index, g→index */
    if (r) return r;
    ⟨ Find f ∧ g recursively 75 ⟩;
  }
```

75. I assume that $f\to lo$ and $f\to hi$ belong to the same octabyte.

The *rmems* counter is incremented only after we've checked for special terminal cases. When none of the simplifications apply, we must prepare to plunge in to deeper waters.

⟨Find $f \wedge g$ recursively 75⟩ ≡

```
rmems++;  /* track recursion overhead */
oo, r0 = and_rec(node_(f->lo), node_(g->lo));
if (!r0) return Λ;  /* oops, trouble */
r1 = and_rec(node_(f->hi), node_(g->hi));
if (!r1) {
  deref(r0);  /* too bad, but we have to abort in midstream */
  return Λ;
}
r = unique_find(vf, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (vf < tvar))
    printf("␣␣␣%x=%x&%x␣(level␣%d)\n", id(r), id(f), id(g), vf − varhead);
  cache_insert(f, g, node_(1), r);
}
return r;
```

This code is used in section 74.

76. With ZDDs, $f \vee g$ is *not* dual to $f \wedge g$, as it was in BDD14.

⟨Subroutines 8⟩ +≡

```
node *or_rec(node *f, node *g)
{
  var *v, *vf, *vg;
  node *r, *r0, *r1;
  if (f ≡ g) return oo, f->xref++, f;  /* f ∨ f = f */
  if (f > g) r = f, f = g, g = r;  /* wow */
  if (f ≡ botsink) return oo, g->xref++, g;  /* 0 ∨ g = g */
  oo, r = cache_lookup(f, g, node_(7));
  if (r) return r;
  ⟨Find f ∨ g recursively 77⟩;
}
```

77. ⟨ Find $f \lor g$ recursively 77 ⟩ ≡

```
rmems++;   /* track recursion overhead */
vf = thevar(f);
vg = thevar(g);
if (vf < vg) {
  v = vf;
  if (o, f ≡ node_(vf→taut)) return oo, f→xref++, f;   /* 1 ∨ g = 1 */
  o, r0 = or_rec(node_(f→lo), g);
  if (¬r0) return Λ;
  r1 = node_(f→hi), oo, r1→xref++;
} else {
  v = vg;
  if (o, g ≡ node_(vg→taut)) return oo, g→xref++, g;   /* f ∨ 1 = 1 */
  if (vg < vf) {
    o, r0 = or_rec(f, node_(g→lo));
    if (¬r0) return Λ;
    r1 = node_(g→hi), oo, r1→xref++;
  } else {
    oo, r0 = or_rec(node_(f→lo), node_(g→lo));
    if (¬r0) return Λ;   /* oops, trouble */
    r1 = or_rec(node_(f→hi), node_(g→hi));
    if (¬r1) {
      deref(r0);   /* too bad, but we have to abort in midstream */
      return Λ;
    }
  }
}
r = unique_find(v, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("   %x=%x|%x (level %d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(7), r);
}
return r;
```

This code is used in section 76.

78. Exclusive or is much the same.

⟨ Subroutines 8 ⟩ +≡

```
node *xor_rec(node *f, node *g)
{
  var *v, *vf, *vg;
  node *r, *r0, *r1;
  if (f ≡ g) return oo, botsink→xref++, botsink;   /* f ⊕ f = 0 */
  if (f > g) r = f, f = g, g = r;   /* wow */
  if (f ≡ botsink) return oo, g→xref++, g;   /* 0 ⊕ g = g */
  oo, r = cache_lookup(f, g, node_(6));
  if (r) return r;
  ⟨ Find f ⊕ g recursively 79 ⟩;
}
```

79. ⟨ Find $f \oplus g$ recursively 79 ⟩ ≡

```
rmems++;    /* track recursion overhead */
vf = thevar(f);
vg = thevar(g);
if (vf < vg) {
  v = vf;
  o, r0 = xor_rec(node_(f→lo), g);
  if (¬r0) return Λ;
  r1 = node_(f→hi), oo, r1→xref++;
} else {
  v = vg;
  if (vg < vf) {
    o, r0 = xor_rec(f, node_(g→lo));
    if (¬r0) return Λ;
    r1 = node_(g→hi), oo, r1→xref++;
  } else {
    oo, r0 = xor_rec(node_(f→lo), node_(g→lo));
    if (¬r0) return Λ;    /* oops, trouble */
    r1 = xor_rec(node_(f→hi), node_(g→hi));
    if (¬r1) {
      deref(r0);    /* too bad, but we have to abort in midstream */
      return Λ;
    }
  }
}
r = unique_find(v, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x^%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(6), r);
}
return r;
```

This code is used in section 78.

80. ZDDs work well only with "normal" operators $\circ$, namely operators such that $0 \circ 0 = 0$. We've done $\wedge$, $\vee$, and $\oplus$; here's the other one.

⟨ Subroutines 8 ⟩ +≡

```
node *but_not_rec(node *f, node *g)
{
  var *vf, *vg;
  node *r, *r0, *r1;
  if (f ≡ g ∨ f ≡ botsink) return oo, botsink→xref++, botsink;    /* f ∧ f̄ = 0 ∧ f̄ = 0 */
  if (g ≡ botsink) return oo, f→xref++, f;    /* f ∧ 0̄ = f */
  oo, vf = thevar(f), vg = thevar(g);
  while (vg < vf) {
    oo, g = node_(g→lo), vg = thevar(g);
    if (f ≡ g) return oo, botsink→xref++, botsink;
    if (g ≡ botsink) return oo, f→xref++, f;
  }
  r = cache_lookup(f, g, node_(2));
  if (r) return r;
  ⟨ Find f ∧ ḡ recursively 81 ⟩;
}
```

81. ⟨ Find $f \wedge \bar{g}$ recursively 81 ⟩ ≡

```
rmems++;    /* track recursion overhead */
if (vf < vg) {
  o, r0 = but_not_rec(node_(f→lo), g);
  if (¬r0) return Λ;
  r1 = node_(f→hi), oo, r1→xref++;
} else {
  oo, r0 = but_not_rec(node_(f→lo), node_(g→lo));
  if (¬r0) return Λ;    /* oops, trouble */
  r1 = but_not_rec(node_(f→hi), node_(g→hi));
  if (¬r1) {
    deref(r0);    /* too bad, but we have to abort in midstream */
    return Λ;
  }
}
r = unique_find(vf, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (vf < tvar))
    printf("␣␣␣%x=%x>%x␣(level␣%d)\n", id(r), id(f), id(g), vf − varhead);
  cache_insert(f, g, node_(2), r);
}
return r;
```

This code is used in section 80.

82. The product operation $f \sqcup g$ is new in BDD15: It corresponds to $f \sqcup g(z) = \exists x\, \exists y\, ((z = x \vee y) \wedge f(x) \wedge g(y))$. Or, if we think of f and g as representing families of subsets, $f \sqcup g = \{\alpha \cup \beta \mid \alpha \in f, \beta \in g\}$.

In particular, $e_i \sqcup e_j \sqcup e_k$ is the family that contains the single subset $\{e_i, e_j, e_k\}$.

Minato used '$*$' for this operation, so ZDDL calls it '*****'.

```
⟨Subroutines 8⟩ +≡
  node *prod_rec(node *f, node *g)
  {
    var *v, *vf, *vg;
    node *r, *r0, *r1, *r01, *r10;
    if (f > g) r = f, f = g, g = r;      /* wow */
    if (f ≤ topsink) {
      if (f ≡ botsink) return oo, f→xref++, f;      /* 0 ⊔ g = 0 */
      else return oo, g→xref++, g;      /* {∅} ⊔ g = g */
    }
    o, v = vf = thevar(f);
    o, vg = thevar(g);
    if (vf > vg) r = f, f = g, g = r, v = vg;
    r = cache_lookup(f, g, node_(5));
    if (r) return r;
    ⟨Find f ⊔ g recursively 83⟩;
  }
```

83. In this step I compute $g_l \vee g_h$ and join it with f_h, instead of joining g_h with $f_l \vee f_h$. This asymmetry can be a big win, but I suppose it can also be a big loss. (Indeed, the similar choice for *coprod_rec* was a mistake, in the common case $f =$ `c1` for coproduct, so I interchanged the roles of f and g in that routine.)

My previous draft of BDD15 computed the OR of *three* joins; that was symmetrical in f and g, but it ran slower in most of my experiments.

I have no good ideas about how to choose automatically between three competing ways to implement this step.

⟨Find $f \sqcup g$ recursively 83⟩ ≡

```
rmems++;    /* track recursion overhead */
if (vf ≠ vg) {
  o, r0 = prod_rec(node_(f→lo), g);
  if (¬r0) return Λ;
  r1 = prod_rec(node_(f→hi), g);
  if (¬r1) {
    deref(r0);    /* too bad, but we have to abort in midstream */
    return Λ;
  }
} else {
  o, r10 = or_rec(node_(g→lo), node_(g→hi));
  if (¬r10) return Λ;
  o, r = prod_rec(node_(f→hi), r10);
  deref(r10);
  if (¬r) return Λ;
  r01 = prod_rec(node_(f→lo), node_(g→hi));
  if (¬r01) {
    deref(r); return Λ;
  }
  r1 = or_rec(r, r01);
  deref(r); deref(r01);
  if (¬r1) return Λ;
  r0 = prod_rec(node_(f→lo), node_(g→lo));
  if (¬r0) {
    deref(r1); return Λ;
  }
}
r = unique_find(v, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x*%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(5), r);
}
return r;
```

This code is used in section 82.

84. The disproduct operation is similar to product, but it evaluates $\{\alpha\cup\beta \mid \alpha\in f, \beta\in g, \alpha\cup\beta=\emptyset\}$. (In other words, all unions of *disjoint* members of f and g, not all unions of the members.)

It's an experimental function that I haven't seen in the literature; I added it shortly after completing the first draft of Section 7.1.4. I wouldn't be surprised if it has lots of uses. I haven't decided on a notation; maybe $\sqcup$ with an extra vertical line in the middle.

```
⟨ Subroutines 8 ⟩ +≡
  node *disprod_rec(node *f, node *g)
  {
    var *v, *vf, *vg;
    node *r, *r0, *r1, *r01;
    if (f > g) r = f, f = g, g = r;     /* wow */
    if (f ≤ topsink) {
      if (f ≡ botsink) return oo, f→xref++, f;
      else return oo, g→xref++, g;
    }
    o, v = vf = thevar(f);
    o, vg = thevar(g);
    if (vf > vg) r = f, f = g, g = r, v = vg;
    r = cache_lookup(f, g, node_(0));
    if (r) return r;
    ⟨ Find the disjoint f ⊔ g recursively 85 ⟩;
  }
```

85. ⟨ Find the disjoint $f \sqcup g$ recursively 85 ⟩ ≡

```
rmems++;   /* track recursion overhead */
if (vf ≠ vg) {
  o, r0 = disprod_rec(node_(f→lo), g);
  if (¬r0) return Λ;
  r1 = disprod_rec(node_(f→hi), g);
  if (¬r1) {
    deref(r0);   /* too bad, but we have to abort in midstream */
    return Λ;
  }
} else {
  o, r = disprod_rec(node_(f→hi), node_(g→lo));
  if (¬r) return Λ;
  r01 = disprod_rec(node_(f→lo), node_(g→hi));
  if (¬r01) {
    deref(r); return Λ;
  }
  r1 = or_rec(r, r01);
  deref(r); deref(r01);
  if (¬r1) return Λ;
  r0 = disprod_rec(node_(f→lo), node_(g→lo));
  if (¬r0) {
    deref(r1); return Λ;
  }
}
r = unique_find(v, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x+%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(0), r);
}
return r;
```

This code is used in section 84.

86. The coproduct operation $f \sqcap g$, which is analogous to $f \sqcup g$, is defined by the similar rule $f \sqcap g(z) = \exists x\, \exists y\, ((z = x \wedge y) \wedge f(x) \wedge g(y))$. Or, if we think of f and g as representing families of subsets, $f \sqcap g = \{\alpha \cap \beta \mid \alpha \in f, \beta \in g\}$.

I'm not sure how I'll want to use this, if it all. But it does seem to belong. The ZDDL notation is ", for no very good reason.

⟨ Subroutines 8 ⟩ +≡

```
node *coprod_rec(node *f, node *g)
{
  var *v, *vf, *vg;
  node *r, *r0, *r1, *r01, *r10;
  if (f > g) r = f, f = g, g = r;   /* wow */
  if (f ≤ topsink) return oo, f→xref++, f;   /* 0 ⊓ g = 0, and {∅} ⊓ g = {∅} when g ≠ 0 */
  oo, r = cache_lookup(f, g, node_(8));
  if (r) return r;
  ⟨ Find f ⊓ g recursively 87 ⟩;
}
```

87. ⟨ Find $f \sqcap g$ recursively 87 ⟩ ≡

```
rmems++;   /* track recursion overhead */
v = vf = thevar(f), vg = thevar(g);
if (vf ≠ vg) {
  if (vf > vg) r = f, f = g, g = r;
  o, r0 = or_rec(node_(f→lo), node_(f→hi));
  if (¬r0) return Λ;
  r = coprod_rec(r0, g);   /* tail recursion won't quite work here */
  deref(r0);   /* (because r0 needs to be dereffed after use) */
} else {
  o, r10 = or_rec(node_(f→lo), node_(f→hi));
  if (¬r10) return Λ;
  o, r = coprod_rec(r10, node_(g→lo));
  deref(r10);
  if (¬r) return Λ;
  r01 = coprod_rec(node_(f→lo), node_(g→hi));
  if (¬r01) {
    deref(r); return Λ;
  }
  r0 = or_rec(r, r01);
  deref(r); deref(r01);
  if (¬r0) return Λ;
  r1 = coprod_rec(node_(f→hi), node_(g→hi));
  if (¬r1) {
    deref(r1); return Λ;
  }
  r = unique_find(v, r0, r1);
}
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x_%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(8), r);
}
return r;
```

This code is used in section 86.

88. Similarly, there's a delta operation $f \mathbin{\Delta} g = \exists x\, \exists y\, ((z = x \oplus y) \land f(x) \land g(y))$. Or, if we think of f and g as representing families of subsets, $f \mathbin{\Delta} g = \{\alpha \mathbin{\Delta} \beta \mid \alpha \in f, \beta \in g\}$.

In ZDDL I use the symbol _, thinking of complementation.

```
⟨Subroutines 8⟩ +≡
  node *delta_rec(node *f, node *g)
  {
    var *v, *vf, *vg;
    node *r, *r0, *r1, *r00, *r01, *r10, *r11;
    if (f > g) r = f, f = g, g = r;      /* wow */
    if (f ≤ topsink) {
      if (f ≡ botsink) return oo, f→xref++, f;      /* 0 Δ g = 0 */
      else return oo, g→xref++, g;      /* {∅} Δ g = g */
    }
    o, v = vf = thevar(f);
    o, vg = thevar(g);
    if (vf > vg) r = f, f = g, g = r, v = vg;
    r = cache_lookup(f, g, node_(11));
    if (r) return r;
    ⟨Find f Δ g recursively 89⟩;
  }
```

89. ⟨ Find $f \,\Delta\, g$ recursively 89 ⟩ ≡

```
rmems++;   /* track recursion overhead */
if (vf ≠ vg) {
  o, r0 = delta_rec(node_(f→lo), g);
  if (¬r0) return Λ;
  r1 = delta_rec(node_(f→hi), g);
  if (¬r1) {
    deref(r0);   /* too bad, but we have to abort in midstream */
    return Λ;
  }
} else {
  oo, r01 = delta_rec(node_(f→lo), node_(g→hi));
  if (¬r01) return Λ;
  r10 = delta_rec(node_(f→hi), node_(g→lo));
  if (¬r10) {
    deref(r01); return Λ;
  }
  r1 = or_rec(r01, r10);
  deref(r01); deref(r10);
  if (¬r1) return Λ;
  r11 = delta_rec(node_(f→hi), node_(g→hi));
  if (¬r11) {
    deref(r1); return Λ;
  }
  r00 = delta_rec(node_(f→lo), node_(g→lo));
  if (¬r00) {
    deref(r1); deref(r11); return Λ;
  }
  r0 = or_rec(r00, r11);
  deref(r00); deref(r11);
  if (¬r0) {
    deref(r1); return Λ;
  }
}
r = unique_find(v, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x#%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(11), r);
}
return r;
```

This code is used in section 88.

90. The quotient and remainder operations have a somewhat different sort of recursion, and I don't know how slow they will be in the worst cases. In common cases, though, they are nice and fast.

The quotient f/g is the family of all subsets α such that, for all $\beta \in g$, $\alpha \cap \beta = \emptyset$ and $\alpha \cup \beta \in f$. (In particular, 0/0 turns out to be 1, the family of *all* subsets.)

The remainder $f \bmod g$ is $f \setminus ((f/g) \sqcup g)$.

In the simplest cases, g is just e_i. Then $f = f_0 \vee (e_i \sqcup f_1)$, where $f_0 = f \bmod e_i$ and $f_1 = f/e_i$. These are the ZDD branches at the root of f, if f is rooted at variable i. I implement these two cases first.

```
⟨ Subroutines 8 ⟩ +≡
  node *ezrem_rec(node *f, var *vg)
  {
    var *vf;
    node *r, *r0, *r1;
    o, vf = thevar(f);
    if (vf ≡ vg) {
      r = node_(f→lo);
      return oo, r→xref ++, r;
    }
    if (vf > vg) return oo, f→xref ++, f;
    o, r = cache_lookup(f, node_(vg→elt), node_(10));
    if (r) return r;
    ⟨ Find f mod g recursively 91 ⟩;
  }
```

91. ⟨ Find f mod g recursively 91 ⟩ ≡

```
  rmems ++;
  o, r0 = ezrem_rec(node_(f→lo), vg);
  if (¬r0) return Λ;
  r1 = ezrem_rec(node_(f→hi), vg);
  if (¬r1) {
    deref(r0); return Λ;
  }
  r = unique_find(vf, r0, r1);
  if (r) {
    if ((verbose & 128) ∧ (vf < tvar))
      printf("␣␣␣%x=%x%%%x␣(level␣%d)\n", id(r), id(f), id(vg→elt), vf − varhead);
    cache_insert(f, node_(vg→elt), node_(10), r);
  }
  return r;
```

This code is used in section 90.

92. ⟨ Subroutines 8 ⟩ +≡

```
node *ezquot_rec(node *f, var *vg)
{
  var *vf;
  node *r, *r0, *r1;
  o, vf = thevar(f);
  if (vf ≡ vg) {
    r = node_(f→hi);
    return oo, r→xref++, r;
  }
  if (vf > vg) return oo, botsink→xref++, botsink;
  o, r = cache_lookup(f, node_(vg→elt), node_(9));
  if (r) return r;
  ⟨ Find f/g recursively in the easy case 93 ⟩;
}
```

93. ⟨ Find f/g recursively in the easy case 93 ⟩ ≡

```
rmems++;
o, r0 = ezquot_rec(node_(f→lo), vg);
if (¬r0) return Λ;
r1 = ezquot_rec(node_(f→hi), vg);
if (¬r1) {
  deref(r0); return Λ;
}
r = unique_find(vf, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (vf < tvar))
    printf("␣␣␣%x=%x%%%x␣(level␣%d)\n", id(r), id(f), id(vg→elt), vf − varhead);
  cache_insert(f, node_(vg→elt), node_(9), r);
}
return r;
```

This code is used in section 92.

94. Now for the general case of division, which also simplifies in several other ways. (This algorithm is due to Shin-ichi Minato, 1994.)

⟨ Subroutines 8 ⟩ +≡

```
node *quot_rec(node *f, node *g)
{
  node *r, *r0, *r1, *f0, *f1;
  var *vf, *vg;
  if (g ≤ topsink) {
    if (g ≡ topsink) return oo, f→xref++, f;     /* f/{∅} = f */
    return oo, tautology→xref++, tautology;     /* f/0 = 1 */
  }
  if (f ≤ topsink) return oo, botsink→xref++, botsink;
  if (f ≡ g) return oo, topsink→xref++, topsink;
  if (o, node_(g→lo) ≡ botsink ∧ node_(g→hi) ≡ topsink) return o, ezquot_rec(f, thevar(g));
  r = cache_lookup(f, g, node_(9));
  if (r) return r;
  ⟨ Find f/g recursively in the general case 95 ⟩;
}
```

95. ⟨Find f/g recursively in the general case 95⟩ ≡

```
rmems++;
o, vg = thevar(g);
f1 = ezquot_rec(f, vg);
if (¬f1) return Λ;
r = quot_rec(f1, node_(g→hi));
deref(f1);
if (¬r) return Λ;
if (r ≠ botsink ∧ node_(g→lo) ≠ botsink) {
  r1 = r;
  f0 = ezrem_rec(f, vg);
  if (¬f0) return Λ;
  r0 = quot_rec(f0, node_(g→lo));
  deref(f0);
  if (¬r0) {
    deref(r1); return Λ;
  }
  r = and_rec(r1, r0);
  deref(r1); deref(r0);
}
if (r) {
  if ((verbose & 128) ∧ (vg < tvar))
    printf("␣␣␣%x=%x/%x␣(level␣%d)\n", id(r), id(f), id(g), vg − varhead);
  cache_insert(f, g, node_(9), r);
}
return r;
```

This code is used in section 94.

96. At present, I don't look for any special cases of the remainder operation except the "ezrem" case. Everything else is done the hard way.

⟨ Subroutines 8 ⟩ +≡

```
node *rem_rec(node *f, node *g)
{
  node *r, *r1;
  var *vf;
  if (g ≤ topsink) {
    if (g ≡ botsink) return oo, f→xref++, f;   /* f mod ∅ = f */
    return oo, botsink→xref++, botsink;   /* f mod {∅} = ∅ */
  }
  if (o, node_(g→lo) ≡ botsink ∧ node_(g→hi) ≡ topsink) return o, ezrem_rec(f, thevar(g));
  r = cache_lookup(f, g, node_(10));
  if (r) return r;
  r = quot_rec(f, g);
  if (¬r) return Λ;
  r1 = prod_rec(r, g);
  deref(r);
  if (¬r1) return Λ;
  r = but_not_rec(f, r1);
  deref(r1);
  if (r) {
    vf = thevar(f);   /* needed only for diagnostics */
    if ((verbose & 128) ∧ (vf < tvar))
      printf("␣␣␣%x=%x%%%x␣(level␣%d)\n", id(r), id(f), id(g), vf − varhead);
    cache_insert(f, g, node_(10), r);
  }
  return r;
}
```

97. Ternary operations. All operations can be reduced to binary operations, but it should be interesting to see if we get a speedup by staying ternary.

I like to call the first one "mux," although many other authors have favored "ite" (meaning if-then-else). The latter doesn't seem right to me when I try to pronounce it. So I'm sticking with the well-worn, traditional name for this function.

The special case $h = 1$ gives "f implies g"; this is a non-normal binary operator, but we still can handle it because ternary mux is normal.

⟨ Subroutines 8 ⟩ +≡

```
node *mux_rec(node *f, node *g, node *h)
{
  var *v, *vf, *vg, *vh;
  node *r, *r0, *r1;
  if (f ≡ botsink) return oo, h→xref++, h;   /* (0? g: h) = h */
  if (g ≡ botsink) return but_not_rec(h, f);   /* (f? 0: h) = h ∧ f̄ */
  if (h ≡ botsink ∨ f ≡ h) return and_rec(f, g);   /* (f? g: f) = (f? g: 0) = f ∧ g */
  if (f ≡ g) return or_rec(f, h);   /* (f? f: h) = f ∨ h */
  if (g ≡ h) return oo, g→xref++, g;   /* (f? g: g) = g */
  ooo, vf = thevar(f), vg = thevar(g), vh = thevar(h);
gloop: while (vg < vf ∧ vg < vh) {
    oo, g = node_(g→lo), vg = thevar(g);
    if (g ≡ botsink) return but_not_rec(h, f);
    if (f ≡ g) return or_rec(f, h);
    if (g ≡ h) return oo, g→xref++, g;
  }
  while (vf < vg ∧ vf < vh) {
    oo, f = node_(f→lo), vf = thevar(f);
    if (f ≡ botsink) return oo, h→xref++, h;
    if (f ≡ h) return and_rec(f, g);
    if (f ≡ g) return or_rec(f, h);   /* (f? f: h) = f ∨ h */
  }
  if (vg < vf ∧ vg < vh) goto gloop;
  if (vf < vg) v = vf; else v = vg;
  if (vh < v) v = vh;
  if (f ≡ node_(v→taut)) return oo, g→xref++, g;   /* (1? g: h) = g */
  if (g ≡ node_(v→taut)) return or_rec(f, h);   /* (f? 1: h) = f ∨ h */
  r = cache_lookup(f, g, h);
  if (r) return r;
  ⟨ Find (f? g: h) recursively 98 ⟩;
}
```

98. ⟨ Find $(f?\ g:\ h)$ recursively 98 ⟩ ≡

```
rmems++;     /* track recursion overhead */
if (v < vf) {     /* in this case v = vh */
  o, r0 = mux_rec(f, (vg ≡ v ? o, node_(g→lo) : g), node_(h→lo));
  if (¬r0) return Λ;     /* oops, trouble */
  r1 = node_(h→hi), oo, r1→xref++;
}
else {     /* in this case v = vg or v = vh */
  o, r0 = mux_rec(node_(f→lo), (vg ≡ v ? o, node_(g→lo) : g), (vh ≡ v ? o, node_(h→lo) : h));
  if (¬r0) return Λ;     /* oops, trouble */
  o, r1 = mux_rec(node_(f→hi), (vg ≡ v ? o, node_(g→hi) : botsink), (vh ≡ v ? o, node_(h→hi) : botsink));
  if (¬r1) {
    deref(r0);     /* too bad, but we have to abort in midstream */
    return Λ;
  }
}
r = unique_find(v, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x?%x:%x␣(level␣%d)\n", id(r), id(f), id(g), id(h), v − varhead);
  cache_insert(f, g, h, r);
}
return r;
```

This code is used in section 97.

99. The median (or majority) operation $\langle fgh\rangle$ has lots of nice symmetry.

⟨ Subroutines 8 ⟩ +≡

```
node *med_rec(node *f, node *g, node *h)
{
  var *v, *vf, *vg, *vh;
  node *r, *r0, *r1;
  ooo, vf = thevar(f), vg = thevar(g), vh = thevar(h);
gloop: if (vg < vf ∨ (vg ≡ vf ∧ g < f)) v = vg, vg = vf, vf = v, r = f, f = g, g = r;
  if (vh < vg ∨ (vh ≡ vg ∧ h < g)) v = vh, vh = vg, vg = v, r = g, g = h, h = r;
  if (vg < vf ∨ (vg ≡ vf ∧ g < f)) v = vg, vg = vf, vf = v, r = f, f = g, g = r;
  if (h ≡ botsink) return and_rec(f, g);     /* ⟨fg0⟩ = f ∧ g */
  if (f ≡ g) return oo, f→xref++, f;     /* ⟨ffh⟩ = f */
  if (g ≡ h) return oo, g→xref++, g;     /* ⟨fgg⟩ = g */
  if (vf < vg) {
    do {
      oo, f = node_(f→lo), vf = thevar(f);
    } while (vf < vg);
    goto gloop;
  }
  r = cache_lookup(f, g, node_(addr_(h) + 1));
  if (r) return r;
  ⟨ Find ⟨fgh⟩ recursively 100 ⟩;
}
```

100. ⟨ Find ⟨fgh⟩ recursively 100 ⟩ ≡

```
rmems++;    /* track recursion overhead */
oo, r0 = med_rec(node_(f→lo), node_(g→lo), (vh ≡ vf ? o, node_(h→lo) : h));
if (¬r0) return Λ;    /* oops, trouble */
if (vf < vh) r1 = and_rec(node_(f→hi), node_(g→hi));
else r1 = med_rec(node_(f→hi), node_(g→hi), node_(h→hi));
if (¬r1) {
  deref(r0);    /* too bad, but we have to abort in midstream */
  return Λ;
}
r = unique_find(vf, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (vf < tvar))
    printf("␣␣␣%x=%x.%x.%x␣(level␣%d)\n", id(r), id(f), id(g), id(h), vf − varhead);
  cache_insert(f, g, node_(addr_(h) + 1), r);
}
return r;
```

This code is used in section 99.

101. More symmetry here.

⟨ Subroutines 8 ⟩ +≡

```
node *and_and_rec(node *f, node *g, node *h)
{
  var *v, *vf, *vg, *vh;
  node *r, *r0, *r1;
  ooo, vf = thevar(f), vg = thevar(g), vh = thevar(h);
restart: while (vf ≠ vg) {
    if (vf < vg) {
      if (g ≡ botsink) return oo, g→xref++, g;
      oo, f = node_(f→lo), vf = thevar(f);    /* wow */
    }
    else if (f ≡ botsink) return oo, f→xref++, f;
    else oo, g = node_(g→lo), vg = thevar(g);
  }
  if (f ≡ g) return and_rec(g, h);    /* f ∧ f ∧ h = f ∧ h */
  while (vf ≠ vh) {
    if (vf < vh) {
      if (h ≡ botsink) return oo, h→xref++, h;
      oooo, f = node_(f→lo), vf = thevar(f), g = node_(g→lo), vg = thevar(g);
      goto restart;
    }
    else oo, h = node_(h→lo), vh = thevar(h);
  }
  if (f > g) {
    if (g > h) r = f, f = h, h = r;
    else if (f > h) r = f, f = g, g = h, h = r;
    else r = f, f = g, g = r;
  } else if (g > h) {
    if (f > h) r = f, f = h, h = g, g = r;
    else r = g, g = h, h = r;
  }    /* now f ≤ g ≤ h */
  if (f ≡ g) return and_rec(g, h);    /* f ∧ f ∧ h = f ∧ h */
  if (g ≡ h) return and_rec(f, g);    /* f ∧ g ∧ g = f ∧ g */
  if (o, f ≡ node_(vf→taut)) return and_rec(g, h);    /* 1 ∧ g ∧ h = g ∧ h */
  if (g ≡ node_(vf→taut)) return and_rec(f, h);
  if (h ≡ node_(vf→taut)) return and_rec(f, g);
  r = cache_lookup(f, g, node_(addr_(h) + 2));
  if (r) return r;
  ⟨ Find f ∧ g ∧ h recursively 102 ⟩;
}
```

102. ⟨ Find $f \wedge g \wedge h$ recursively 102 ⟩ ≡

```
rmems++;    /* track recursion overhead */
ooo, r0 = and_and_rec(node_(f→lo), node_(g→lo), node_(h→lo));
if (¬r0) return Λ;    /* oops, trouble */
r1 = and_and_rec(node_(f→hi), node_(g→hi), node_(h→hi));
if (¬r1) {
  deref(r0);    /* too bad, but we have to abort in midstream */
  return Λ;
}
r = unique_find(vf, r0, r1);
if (r) {
  if ((verbose & 128) ∧ (vf < tvar))
    printf("␣␣␣%x=%x&%x&%x␣(level␣%d)\n", id(r), id(f), id(g), id(h), vf − varhead);
  cache_insert(f, g, node_(addr_(h) + 2), r);
}
return r;
```

This code is used in section 101.

103. The *symfunc* operation is a ternary relation of a different kind: Its first parameter is a node, its second parameter is a variable, and its third parameter is an integer.

More precisely, *symfunc* has the following three arguments: First, p specifies a list of t variables, ideally in the form $e_{i_1} \vee \cdots \vee e_{i_t}$ for some $t \geq 0$. (However, the exact form of p is not checked; the sequence of LO pointers defines the actual list.) Second, v is a variable; and k is an integer. The meaning is to return the function that is true if and only if exactly k of the listed variables $\geq v$ are true and all variables $< v$ are false. For example, $symfunc(e_1 \vee e_4 \vee e_6, varhead + 2, 2)$ is the ZDD for $\bar{x}_0 \wedge \bar{x}_1 \wedge S_2(x_4, x_6)$.

Beware: If parameter p doesn't have the stated "ideal" form, reordering of variables can screw things up.

⟨ Subroutines 8 ⟩ +≡

```
node *symfunc(node *p, var *v, int k)
{
  register var *vp;
  register node *q, *r;
  o, vp = thevar(p);
  while (vp < v) oo, p = node_(p→lo), vp = thevar(p);
  if (vp ≡ topofvars) {      /* empty list */
    if (k > 0) return oo, botsink→xref++, botsink;
    else return oo, node_(v→taut)→xref++, node_(v→taut);
  }
  oooo, r = cache_lookup(p, node_(v→taut), node_(varhead[k].taut + 4));
  if (r) return r;
  rmems++;
  o, q = symfunc(node_(p→lo), vp + 1, k);
  if (¬q) return Λ;
  if (k > 0) {
    r = symfunc(node_(p→lo), vp + 1, k − 1);
    if (¬r) {
      deref(q);
      return Λ;
    }
    q = unique_find(vp, q, r);
    if (¬q) return Λ;
  }
  while (vp > v) {
    vp−−;
    oo, q→xref++;
    q = unique_find(vp, q, q);
    if (¬q) return Λ;
  }
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x@%x@%x␣(level␣%d)\n", id(q), id(p), id(v→taut), id(varhead[k].taut), v − varhead);
  cache_insert(p, node_(v→taut), node_(varhead[k].taut + 4), q);
  return q;
}
```

104. There's also a kludgy ternary operation intended for building arbitrary ZDDs from the bottom up. Namely, $f!\ g{:}\ h$ returns a ZDD node that branches on x_i, with g and h as the lo and hi pointers, provided that $f = e_i$ and that the roots of g and h are greater than x_i. (For any other values of f, g, and h, we just do something that runs to completion without screwing up.)

⟨ Subroutines 8 ⟩ +≡

```
node *zdd_build(node *f, node *g, node *h)
{
  var *vf;
  node *r;
  if (f ≤ topsink) return oo, f→xref++, f;
  o, vf = thevar(f);
  while ((o, thevar(g)) ≤ vf) g = node_(g→lo);
  while ((o, thevar(h)) ≤ vf) h = node_(h→lo);
  oooo, g→xref++, h→xref++;
  r = unique_find(vf, g, h);
  if (r) {
    if ((verbose & 128) ∧ (vf < tvar))
      printf("␣␣␣%x=%x!%x:%x␣(level␣%d)\n", id(r), id(f), id(g), id(h), vf − varhead);
  }
  return r;
}
```

105. Top-level calls. As mentioned above, there's a top-level "wrapper" around each of the recursive synthesis routines, so that we can launch them properly.

Here's the top-level routine for binary operators.

⟨Subroutines 8⟩ +≡
```
node *binary_top(int curop, node *f, node *g)
{
  node *r;
  unsigned long long oldmems = mems, oldrmems = rmems, oldzmems = zmems;
  if (verbose & 2) printf("beginning␣to␣compute␣%x␣%s␣%x:\n", id(f), binopname[curop], id(g));
  cacheinserts = 0;
  while (1) {
    switch (curop) {
    case 0: r = disprod_rec(f, g); break;   /* disjoint variant of f ⊔ g */
    case 1: r = and_rec(f, g); break;   /* f ∧ g */
    case 2: r = but_not_rec(f, g); break;   /* f ∧ ḡ */
    case 4: r = but_not_rec(g, f); break;   /* f̄ ∧ g */
    case 5: r = prod_rec(f, g); break;   /* f ⊔ g */
    case 6: r = xor_rec(f, g); break;   /* f ⊕ g */
    case 7: r = or_rec(f, g); break;   /* f ∨ g */
    case 8: r = coprod_rec(f, g); break;   /* f ⊓ g */
    case 9: r = quot_rec(f, g); break;   /* f/g */
    case 10: r = rem_rec(f, g); break;   /* f mod g */
    case 11: r = delta_rec(f, g); break;   /* f Δ g */
    default: fprintf(stderr, "This␣can't␣happen!\n"); exit(−69);
    }
    if (r) break;
    attempt_repairs();   /* try to carry on */
  }
  if (verbose & (1 + 2)) printf("␣%x=%x%s%x␣(%llu␣mems,␣%llu␣rmems,␣%llu␣zmems,␣%.4g)\n",
        id(r), id(f), binopname[curop], id(g), mems − oldmems, rmems − oldrmems, zmems − oldzmems,
        mems − oldmems + rfactor * (rmems − oldrmems) + zfactor * (zmems − oldzmems));
  return r;
}
```

106. ⟨Templates for subroutines 26⟩ +≡
```
void attempt_repairs(void);   /* collect garbage or something if there's hope */
```

107. ⟨ Subroutines 8 ⟩ +≡

```
node *ternary_top(int curop, node *f, node *g, node *h)
{
  node *r;
  unsigned long long oldmems = mems, oldrmems = rmems, oldzmems = zmems;
  if (verbose & 2) printf("beginning to compute %x %s %x %s %x:\n", id(f),
        ternopname1[curop - 16], id(g), ternopname2[curop - 16], id(h));
  cacheinserts = 0;
  while (1) {
    switch (curop) {
    case 16: r = mux_rec(f, g, h); break;    /* f? g: h */
    case 17: r = med_rec(f, g, h); break;    /* ⟨fgh⟩ */
    case 18: r = and_and_rec(f, g, h); break;    /* f ∧ g ∧ h */
    case 19: r = zdd_build(f, g, h); break;    /* f! g: h */
    default: fprintf(stderr, "This can't happen!\n"); exit(-69);
    }
    if (r) break;
    attempt_repairs();    /* try to carry on */
  }
  if (verbose & (1 + 2)) printf(" %x=%x%s%x%s%x (%llu mems, %llu rmems, %llu zmems, %.4g)\n",
        id(r), id(f), ternopname1[curop - 16], id(g), ternopname2[curop - 16], id(h), mems - oldmems,
        rmems - oldrmems, zmems - oldzmems,
        mems - oldmems + rfactor * (rmems - oldrmems) + zfactor * (zmems - oldzmems));
  return r;
}
node *symfunc_top(node *p, int k)
{
  node *r;
  unsigned long long oldmems = mems, oldrmems = rmems, oldzmems = zmems;
  if (verbose & 2) printf("beginning to compute %x S %d:\n", id(p), k);
  cacheinserts = 0;
  while (1) {
    r = symfunc(p, varhead, k);
    if (r) break;
    attempt_repairs();    /* try to carry on */
  }
  if (verbose & (1 + 2)) printf(" %x=%xS%d (%llu mems, %llu rmems, %llu zmems, %.4g)\n",
        id(r), id(p), k, mems - oldmems, rmems - oldrmems, zmems - oldzmems,
        mems - oldmems + rfactor * (rmems - oldrmems) + zfactor * (zmems - oldzmems));
  return r;
}
```

108. Parsing the commands. We're almost done, but we need to control the overall process by obeying the user's instructions. The syntax for elementary user commands appeared at the beginning of this program; now we want to flesh it out and implement it.

⟨ Read a command and obey it; **goto** *alldone* if done 108 ⟩ ≡

```
{
  ⟨ Make sure the coast is clear 109 ⟩;
  ⟨ Fill buf with the next command, or goto alldone 111 ⟩;
  ⟨ Parse the command and execute it 112 ⟩;
}
```

This code is used in section 3.

109. Before we do any commands, it's helpful to ensure that no embarrassing anomalies will arise.

```
#define debugging 1
```

⟨ Make sure the coast is clear 109 ⟩ ≡

```
#if debugging & includesanity
  if (verbose & 8192) sanity_check( );
#endif
  if (totalnodes ≥ toobig) ⟨ Invoke autosifting 151 ⟩;
  if (verbose & 1024) show_stats( );
```

This code is used in section 108.

110.

```
#define bufsize 100   /* all commands are very short, but comments might be long */
```

⟨ Global variables 5 ⟩ +≡

```
  char buf[bufsize];   /* our master's voice */
```

111. ⟨ Fill *buf* with the next command, or **goto** *alldone* 111 ⟩ ≡

```
  if (infile) {
    if (¬fgets(buf, bufsize, infile)) {   /* assume end of file */
      if (file_given) goto alldone;   /* quit the program if the file was argv[1] */
      fclose(infile);
      infile = Λ;
      continue;
    }
    if (verbose & 64) printf(">␣%s", buf);
  } else while (1) {
      printf(">␣"); fflush(stdout);   /* prompt the user */
      if (fgets(buf, bufsize, stdin)) break;
      freopen("/dev/tty", "r", stdin);   /* end of command-line stdin */
    }
```

This code is used in section 108.

112. The first nonblank character of each line identifies the type of command. All-blank lines are ignored; so are lines that begin with '#'.

I haven't attempted to make this interface the slightest bit fancy. Nor have I had time to write a detailed explanation of how to use this program—sorry. Hopefully someone like David Pogue will be motivated to write the missing manual.

```
#define getk for (k = 0; isdigit(*c); c++) k = 10 * k + *c - '0'   /* scan a number */
#define reporterror
    { printf("Sorry; '%c' confuses me %s%s",
        *(c - 1), infile ? "in this command: " : "in that command.", infile ? buf : "\n");
      goto nextcommand; }
```

⟨ Parse the command and execute it 112 ⟩ ≡

```
rescan: for (c = buf; *c ≡ ' '; c++) ;   /* pass over initial blanks */
  switch (*c++) {
  case '\n': if (¬infile) printf("(Type 'quit' to exit the program.)\n");
  case '#': continue;
  case '!': printf(buf + 1); continue;   /* echo the input line on stdout */
  case 'b': ⟨ Bubble sort to reestablish the natural variable order 144 ⟩; continue;
  case 'C': print_cache(); continue;
  case 'f': ⟨ Parse and execute an assignment to f_k 118 ⟩; continue;
  case 'i': ⟨ Get ready to read a new input file 114 ⟩; continue;
  case 'l': getk; leasesonlife = k; continue;
  case 'm': ⟨ Print a Mathematica program for a generating function 157 ⟩; continue;
  case 'o': ⟨ Output a function 116 ⟩; continue;
  case 'O': ⟨ Print the current variable ordering 117 ⟩; continue;
  case 'p': ⟨ Print a function or its profile 115 ⟩; continue;
  case 'P': print_base(0); continue;   /* P means "print all" */
  case 'q': goto alldone;   /* this will exit the program */
  case 'r': ⟨ Reset the reorder trigger 150 ⟩; continue;
  case 's': ⟨ Swap variable x_k with its predecessor 128 ⟩; continue;
  case 'S':
    if (isdigit(*c)) ⟨ Sift on variable x_k 145 ⟩
    else siftall(); continue;
  case 't': ⟨ Reset tvar 127 ⟩; continue;
  case 'v': getk; verbose = k; continue;
  case 'V': verbose = -1; continue;
  case 'x':
    if (¬totvars) { getk; createvars(k);
    } else reporterror; continue;
  case '$': show_stats(); continue;
  default: reporterror;
  }
nextcommand: continue;
```

This code is used in section 108.

113. ⟨ Local variables 19 ⟩ +≡

```
  char *c, *cc;   /* characters being scanned */
  node *p, *q, *r;   /* operands */
  var *v;   /* a variable */
  int lhs;   /* index on left side of equation */
  int curop;   /* current operator */
```

114. The ⟨special⟩ command `include` ⟨filename⟩ starts up a new infile. (Instead of `include`, you could also say `input` or `i`, or even `ignore`.)

```
#define passblanks for ( ; *c ≡ '␣'; c++)

⟨Get ready to read a new input file 114⟩ ≡
  if (infile) printf("Sorry␣---␣you␣can't␣include␣one␣file␣inside␣of␣another.\n");
  else {
    for ( ; isgraph(*c); c++) ;   /* pass nonblanks */
    passblanks;
    for (cc = c; isgraph(*c); c++) ;   /* pass nonblanks */
    *c = '\0';
    if (¬(infile = fopen(cc, "r"))) printf("Sorry␣---␣I␣couldn't␣open␣file␣`%s'!\n", cc);
  }
```

This code is used in section 112.

115. The command '`p3`' prints out the ZDD for f_3; the command '`pp3`' prints just the profile.

```
#define getkf getk; if (k ≥ extsize) { printf("f%d␣is␣out␣of␣range.\n", k); continue; }
#define getkv getk; if (k ≥ totvars) { printf("x%d␣is␣out␣of␣range.\n", k); continue; }

⟨Print a function or its profile 115⟩ ≡
  if (*c ≡ 'p') {   /* pp means "print a profile" */
    c++; getkf;
    printf("p%d:", k);
    print_profile(f[k]);
  } else {
    getkf;
    printf("f%d=", k);
    print_function(f[k]);
  }
```

This code is used in section 112.

116. ⟨Output a function 116⟩ ≡

```
  getkf;
  sprintf(buf, "/tmp/f%d.zdd", k);
  freopen(buf, "w", stdout);   /* redirect stdout to a file */
  print_function(f[k]);
  freopen("/dev/tty", "w", stdout);   /* restore normal stdout */
```

This code is used in section 112.

117. ⟨Print the current variable ordering 117⟩ ≡

```
  for (v = varhead; v < topofvars; v++) printf("␣x%d", v→name);
  printf("\n");
```

This code is used in section 112.

118. My little finite-state automaton.

⟨ Parse and execute an assignment to f_k 118 ⟩ ≡

```
getkf; lhs = k;
passblanks;
if (*c++ ≠ '=') reporterror;
⟨ Get the first operand, p 119 ⟩;
⟨ Get the operator, curop 121 ⟩;
second: ⟨ Get the second operand, q 122 ⟩;
third: ⟨ If the operator is ternary, get the third operand, r 123 ⟩;
fourth: ⟨ Evaluate the right-hand side and put the answer in r 124 ⟩;
assignit: ⟨ Assign r to f_k, where k = lhs 125 ⟩;
```

This code is used in section 112.

119. #define *checknull*(p)

```
if (¬p) { printf("f%d␣is␣null!\n", k); continue; }
```

⟨ Get the first operand, p 119 ⟩ ≡

```
passblanks;
switch (*c++) {
case 'e': getkv; p = node_(varhead[varmap[k]].elt); break;
case 'x': getkv; p = projection(varmap[k]); break;
case 'f': getkf; p = f[k]; checknull(p); break;
case 'c': p = getconst(*c++); if (¬p) reporterror; break;
case '~': p = tautology; curop = 2; goto second;    /* reduce ¬f to 1 ∧ f̄ */
case '.': ⟨ Dereference the left-hand side 126 ⟩; continue;
default: reporterror;
}
```

This code is used in section 118.

120. The user shouldn't access any constants until specifying the number of variables with the `x` command above.

⟨ Subroutines 8 ⟩ +≡

```
node *getconst(int k)
{
  k -= '0';
  if (k < 0 ∨ k > 2) return Λ;
  if (totvars ≡ 0) {
    printf("(Hey,␣I␣don't␣know␣the␣number␣of␣variables␣yet.)\n");
    return Λ;
  }
  if (k ≡ 0) return botsink;
  if (k ≡ 2) return topsink;
  return tautology;
}
```

121. Many of the operations implemented in BDD14 are not present (yet?) in BDD15.

⟨ Get the operator, *curop* 121 ⟩ ≡

```
passblanks;
switch (*c++) {
case '+': curop = 0; break;      /* disproduct */
case '&': curop = 1; break;      /* and */
case '>': curop = 2; break;      /* butnot */
case '<': curop = 4; break;      /* notbut */
case '*': curop = 5; break;      /* product */
case '^': curop = 6; break;      /* xor */
case '|': curop = 7; break;      /* or */
case '"': curop = 8; break;      /* coproduct */
case '/': curop = 9; break;      /* quotient */
case '%': curop = 10; break;      /* remainder */
case '_': curop = 11; break;      /* delta */
case '?': curop = 16; break;      /* if-then-else */
case '.': curop = 17; break;      /* median */
case '!': curop = 19; break;      /* zdd-build */
case '\n': curop = 7, q = p, c--; goto fourth;      /* change unary p to p ∨ p */
case 'S': getk; r = symfunc_top(p, k); goto assignit;      /* special S op */
default: reporterror;
}
```

This code is used in section 118.

122. ⟨ Get the second operand, *q* 122 ⟩ ≡

```
passblanks;
switch (*c++) {
case 'e': getkv; q = node_(varhead[varmap[k]].elt); break;
case 'x': getkv; q = projection(varmap[k]); break;
case 'f': getkf; q = f[k]; checknull(q); break;
case 'c': q = getconst(*c++); if (¬q) reporterror; break;
default: reporterror;
}
```

This code is used in section 118.

123. ⟨ If the operator is ternary, get the third operand, *r* 123 ⟩ ≡

```
passblanks;
if (curop ≡ 1 ∧ *c ≡ '&') curop = 18;      /* and-and */
if (curop ≤ maxbinop) r = Λ;
else {
  if (*c++ ≠ ternopname2[curop − 16][0]) reporterror;
  passblanks;
  switch (*c++) {
  case 'e': getkv; r = node_(varhead[varmap[k]].elt); break;
  case 'x': getkv; r = projection(varmap[k]); break;
  case 'f': getkf; r = f[k]; checknull(r); break;
  case 'c': r = getconst(*c++); if (¬r) reporterror; break;
  default: reporterror;
  }
}
```

This code is used in section 118.

124. We have made sure that all the necessary operands are non-Λ.

$\langle$ Evaluate the right-hand side and put the answer in r 124 $\rangle \equiv$

```
passblanks;
if (*c ≠ '\n' ∧ *c ≠ '#') {      /* comments may follow '#' */
reportjunk: c++;
    reporterror;
}
if (curop ≤ maxbinop) r = binary_top(curop, p, q);
else r = ternary_top(curop, p, q, r);
```

This code is used in section 118.

125. The *sanity_check* routine tells me that I don't need to increase *r→xref* here (although I'm not sure that I totally understand why).

$\langle$ Assign r to f_k, where $k = lhs$ 125 $\rangle \equiv$

```
if (o, f[lhs]) deref(f[lhs]);
o, f[lhs] = r;
```

This code is used in section 118.

126. $\langle$ Dereference the left-hand side 126 $\rangle \equiv$

```
if (o, f[lhs]) {
    deref(f[lhs]);
    o, f[lhs] = Λ;
}
```

This code is used in section 119.

127. In a long calculation, it's nice to get progress reports by setting bit 128 of the *verbose* switch. But we want to see such reports only near the top of the ZDDs. (Note that *varmap* is not relevant here.)

$\langle$ Reset *tvar* 127 $\rangle \equiv$

```
getkv;
tvar = &varhead[k + 1];
```

This code is used in section 112.

128. Reordering. All of the algorithms for changing the order of variables in a ZDD base are based on a primitive swap-in-place operation, which is made available to the user as an '`s`' command for online experimentation.

The swap-in-place algorithm interchanges $x_u \leftrightarrow x_v$ in the ordering, where x_u immediately precedes x_v. No new dead nodes are introduced during this process, although some nodes will disappear and others will be created. Furthermore, no pointers will change except within nodes that branch on x_u or x_v; every node on level u or level v that is accessible either externally or from above will therefore continue to represent the same subfunction, but in a different way.

⟨ Swap variable x_k with its predecessor 128 ⟩ ≡

```
getkv; v = &varhead[varmap[k]];
reorder_init();    /* prepare for reordering */
if (v→up) swap(v→up, v);
reorder_fin();    /* go back to normal processing */
```

This code is used in section 112.

129. Before we diddle with such a sensitive thing as the order of branching, we must clear the cache. We also remove all dead nodes, which otherwise get in the way. Furthermore, we set the *up* and *down* links inside **var** nodes.

By setting *leasesonlife* = 1 here, I'm taking a rather cowardly approach to the problem of memory overflow: This program will simply give up, when it runs out of elbow room. No doubt there are much better ways to flail about and possibly recover, when memory gets tight, but I don't have the time or motivation to think about them today.

The *up* and *down* fields aren't necessary in BDD15, since $v\to up = v - 1$ and $v\to down = v + 1$ except at the top and bottom. But I decided to save time by simply copying as much code from BDD14 as possible.

⟨ Subroutines 8 ⟩ +≡

```
void reorder_init(void)
{
  var *v, *vup;
  collect_garbage(1);
  totalvars = 0;
  for (v = varhead, vup = Λ; v < topofvars; v++) {
    v→aux = ++totalvars;
    v→up = vup;
    if (vup) vup→down = v; else firstvar = v;
    vup = v;
  }
  if (vup) vup→down = Λ; else firstvar = Λ;
  oldleases = leasesonlife;
  leasesonlife = 1;    /* disallow reservations that fail */
}
void reorder_fin(void)
{
  cache_init();
  leasesonlife = oldleases;
}
```

130. ⟨ Global variables 5 ⟩ +≡

```
int totalvars;    /* this many var records are in use */
var *firstvar;    /* and this one is the smallest in use */
int oldleases;    /* this many "leases on life" have been held over */
```

131. We classify the nodes on levels u and v into four categories: Level-u nodes that branch to at least one level-v node are called "tangled"; the others are "solitary." Level-v nodes that are reachable from levels above u or from external pointers (f_j or x_j or y_j) are called "remote"; the others, which are reachable only from level u, are "hidden."

After the swap, the tangled nodes will remain on level u; but they will now branch on the former x_v, and their *lo* and *hi* pointers will probably change. The solitary nodes will move to level v, where they will become remote; they'll still branch on the former x_u as before. The remote nodes will move to level u, where they will become solitary—still branching as before on the former x_v. The hidden nodes will disappear and be recycled. In their place we might create "newbies," which are new nodes on level v that branch on the old x_u. The newbies are accessible only from tangled nodes that have been transmogrified; hence they will be the hidden nodes, if we decide to swap the levels back again immediately.

Notice that if there are m tangled nodes, there are at most $2m$ hidden nodes, and at most $2m$ newbies. The swap is beneficial if and only if the hidden nodes outnumber the newbies.

The present implementation is based on the assumptions that almost all nodes on level u are tangled and almost all nodes on level v are hidden. Therefore, instead of retaining solitary and remote nodes in their unique tables, deleting the other nodes, swapping unique tables, and then inserting tangled/newbies, we use a different strategy by which both unique tables are essentially trashed and rebuilt from scratch. (In other words, we assume that the deletion of tangled nodes and hidden nodes will cost more than the insertion of solitary nodes and remote nodes.)

We need some way to form temporary lists of all the solitary, tangled, and remote nodes. No link fields are readily available in the nodes themselves, unless we resort to the shadow memory. The present implementation solves the problem by reconfiguring the unique table for level u before destroying it: We move all solitary nodes to the beginning of that table, and all tangled nodes to the end. This approach is consistent with our preference for cache-friendly methods like linear probing.

```
⟨Declare the swap subroutine 131⟩ ≡
  void swap(var *u, var *v)
  {
    register int j, k, solptr, tangptr, umask, vmask, del;
    register int hcount = 0, rcount = 0, scount = 0, tcount = 0, icount = totalnodes;
    register node *f, *g, *h, *gg, *hh, *p, *pl, *ph, *q, *ql, *qh, *firsthidden, *lasthidden;
    register var *vg, *vh;
    unsigned long long omems = mems, ozmems = zmems;

    oo, umask = u→mask, vmask = v→mask;
    del = ((u − varhead) ⊕ (v − varhead)) ≪ (32 − logvarsize);
    ⟨Separate the solitary nodes from the tangled nodes 132⟩;
    ⟨Create a new unique table for x_u and move the remote nodes to it 133⟩;
    if (verbose & 2048)
      printf("swapping␣%d(x%d)<->%d(x%d):␣solitary␣%d,␣tangled␣%d,␣remote␣%d,␣hidden␣%d\n",
          u − varhead, u→name, v − varhead, v→name, scount, tcount, rcount, hcount);
    ⟨Create a new unique table for x_v and move the solitary nodes to it 137⟩;
    ⟨Transmogrify the tangled nodes and insert them in their new guise 138⟩;
    ⟨Delete the lists of solitary, tangled, and hidden nodes 141⟩;
    if (verbose & 2048) printf("␣newbies␣%d,␣change␣%d,␣mems␣(%llu,0,%llu)\n",
          totalnodes − icount + hcount, totalnodes − icount, mems − omems, zmems − ozmems);
    ⟨Swap names and projection functions 142⟩;
  }
```

This code is used in section 143.

132. Here's a cute algorithm something like the inner loop of quicksort. By decreasing the reference counts of the tangled nodes' children, we will be able to distinguish remote nodes from hidden nodes in the next step.

⟨ Separate the solitary nodes from the tangled nodes 132 ⟩ ≡

```
solptr = j = 0; tangptr = k = umask + 1;
while (1) {
  for ( ; j < k; j += sizeof(addr)) {
    oo, p = fetchnode(u, j);
    if (p ≡ 0) continue;
    o, pl = node_(p→lo), ph = node_(p→hi);
    if ((o, thevar(pl) ≡ v) ∨ (o, thevar(ph) ≡ v)) {
      oooo, pl→xref −−, ph→xref −−;
      break;
    }
    storenode(u, solptr, p);
    solptr += sizeof(addr), scount++;
  }
  if (j ≥ k) break;
  for (k −= sizeof(addr); j < k; k −= sizeof(addr)) {
    oo, q = fetchnode(u, k);
    if (q ≡ 0) continue;
    o, ql = node_(q→lo), qh = node_(q→hi);
    if ((o, thevar(ql) ≡ v) ∨ (o, thevar(qh) ≡ v)) oooo, ql→xref −−, qh→xref −−;
    else break;
    tangptr −= sizeof(addr), tcount++;
    storenode(u, tangptr, q);
  }
  tangptr −= sizeof(addr), tcount++;
  storenode(u, tangptr, p);
  if (j ≥ k) break;
  storenode(u, solptr, q);
  solptr += sizeof(addr), scount++;
  j += sizeof(addr);
}
```

This code is used in section 131.

133. We temporarily save the pages of the old unique table, since they now contain the sequential lists of solitary and tangled nodes.

The hidden nodes are linked together by *xref* fields, but not yet recycled (because we will want to look at their *lo* and *hi* fields again).

⟨Create a new unique table for x_u and move the remote nodes to it 133⟩ ≡
```
for (k = 0; k ≤ umask ≫ logpagesize; k++) oo, savebase[k] = u→base[k];
new_unique(u, tcount + 1);   /* initialize an empty unique table */
for (k = rcount = hcount = 0; k < vmask; k += sizeof(addr)) {
  oo, p = fetchnode(v, k);
  if (p ≡ 0) continue;
  if (o, p→xref < 0) {   /* p is a hidden node */
    if (hcount ≡ 0) firsthidden = lasthidden = p, hcount = 1;
    else o, hcount++, p→xref = addr_(lasthidden), lasthidden = p;
    oo, node_(p→lo)→xref−−;   /* recursive euthanization won't be needed */
    oo, node_(p→hi)→xref−−;   /* recursive euthanization won't be needed */
  } else {
    rcount++;   /* p is a remote node */
    oo, p→index ⊕= del;   /* change the level from v to u */
    insert_node(u, p);   /* put it into the new unique table (see below) */
  }
}
```
This code is used in section 131.

134. ⟨Global variables 5⟩ +≡
```
addr savebase[maxhashpages];   /* pages to be discarded after swapping */
```

135. The *new_unique* routine inaugurates an empty unique table with room for at least m nodes before its size will have to double. Those nodes will be inserted soon, so we don't mind that it is initially sparse.

⟨Subroutines 8⟩ +≡
```
void new_unique(var *v, int m)
{
  register int f, j, k;
  for (f = 6; (m ≪ 2) > f; f ≪= 1) ;
  f = f & (−f);
  o, v→free = f, v→mask = (f ≪ 2) − 1;
  for (k = 0; k ≤ v→mask ≫ logpagesize; k++) {
    o, v→base[k] = addr_(reserve_page());   /* it won't be Λ */
    if (k) {
      for (j = v→base[k]; j < v→base[k] + pagesize; j += sizeof(long long)) storenulls(j);
      zmems += pagesize/sizeof(long long);
    }
  }
  f = v→mask & pagemask;
  for (j = v→base[0]; j < v→base[0] + f; j += sizeof(long long)) storenulls(j);
  zmems += (f + 1)/sizeof(long long);
}
```

136. The *insert_node* subroutine is somewhat analogous to *unique_find*, but its parameter q is a node that's known to be unique and not already present. The task is simply to insert this node into the hash table. Complications arise only if the table thereby becomes too full, and needs to be doubled in size, etc.

```
⟨Subroutines 8⟩ +≡
  void insert_node(var *v, node *q)
  {
    register int j, k, mask, free;
    register addr *hash;
    register node *l, *h, *p, *r;
    o, l = node_(q→lo), h = node_(q→hi);
  restart: o, mask = v→mask, free = v→free;
    for (hash = hashcode(l, h); ; hash++) {     /* ye olde linear probing */
      k = addr_(hash) & mask;
      oo, r = fetchnode(v, k);
      if (¬r) break;
    }
    if (−−free ≤ mask ≫ 4) ⟨Double the table size and goto restart 33⟩;
    storenode(v, k, q); o, v→free = free;
    return;
  cramped: printf("Uh␣oh:␣insert_node␣hasn't␣enough␣memory␣to␣continue!\n");
    show_stats();
    exit(−96);
  }
```

137. ⟨Create a new unique table for x_v and move the solitary nodes to it 137⟩ ≡

```
  for (k = 0; k ≤ vmask ≫ logpagesize; k++) o, free_page(page_(v→base[k]));
  new_unique(v, scount);
  for (k = 0; k < solptr; k += sizeof(addr)) {
    o, p = node_(addr__(savebase[k ≫ logpagesize] + (k & pagemask)));
    oo, p→index ⊕= del;     /* change the level from u to v */
    insert_node(v, p);
  }
```

This code is used in section 131.

138. The most dramatic change caused by swapping occurs in this step. Suppose f is a tangled node on level u before the swap, and suppose $g = f{\to}lo$ and $h = f{\to}hi$ are on level v at that time. After swapping, we want $f{\to}lo$ and $f{\to}hi$ to be newbie nodes gg and hh, with $gg{\to}lo = g{\to}lo$, $gg{\to}hi = h{\to}lo$, $hh{\to}lo = g{\to}hi$, $hh{\to}hi = h{\to}hi$. (Actually, gg and hh might not both be newbies, because we might have, say, $h{\to}lo = botsink$.) Similar formulas apply when either g or h lies below level v.

⟨Transmogrify the tangled nodes and insert them in their new guise 138⟩ ≡

```
  for (k = tangptr; k < umask; k += sizeof(addr)) {
    o, f = node_(addr__(savebase[k ≫ logpagesize] + (k & pagemask)));
    o, g = node_(f→lo), h = node_(f→hi);
    oo, vg = thevar(g), vh = thevar(h);     /* N.B.: vg and/or vh might be either u or v at this point */
    gg = swap_find(v, vg > v ? g : (o, node_(g→lo)), vh > v ? h : (o, node_(h→lo)));
    hh = swap_find(v, vg > v ? botsink : node_(g→hi), vh > v ? botsink : node_(h→hi));
    o, f→lo = addr_(gg), f→hi = addr_(hh);     /* (u, gg, hh) will be unique */
    insert_node(u, f);
  }
```

This code is used in section 131.

139. The *swap_find* procedure in the transmogrification step is almost identical to *unique_find*; it differs only in the treatment of reference counts (and the knowledge that no nodes are currently dead).

⟨ Subroutines 8 ⟩ +≡
```
node *swap_find(var *v, node *l, node *h)
{
  register int j, k, mask, free;
  register addr *hash;
  register node *p, *r;
  if (h ≡ botsink) {    /* easy case */
    return oo, l→xref++, l;
  }
restart: o, mask = v→mask, free = v→free;
  for (hash = hashcode(l, h); ; hash++) {    /* ye olde linear probing */
    k = addr_(hash) & mask;
    oo, p = fetchnode(v, k);
    if (¬p) goto newnode;
    if (node_(p→lo) ≡ l ∧ node_(p→hi) ≡ h) break;
  }
  return o, p→xref++, p;
newnode: ⟨ Create a newbie and return it 140 ⟩;
}
```

140. ⟨ Create a newbie and return it 140 ⟩ ≡
```
if (−−free ≤ mask ≫ 4) ⟨ Double the table size and goto restart 33 ⟩;
p = reserve_node();
storenode(v, k, p); o, v→free = free;
initnewnode(p, v − varhead, l, h);
oooo, l→xref++, h→xref++;
return p;
cramped: printf("Uh␣oh:␣swap_find␣hasn't␣enough␣memory␣to␣continue!\n");
show_stats();
exit(−95);
```
This code is used in section 139.

141. ⟨ Delete the lists of solitary, tangled, and hidden nodes 141 ⟩ ≡
```
for (k = 0; k ≤ umask ≫ logpagesize; k++) o, free_page(page_(savebase[k]));
if (hcount) {
  o, firsthidden→xref = addr_(nodeavail);
  nodeavail = lasthidden;
  totalnodes −= hcount;
}
```
This code is used in section 131.

142. All *elt* and *taut* functions are kept internally consistent as if no reordering has taken place. The *varmap* and *name* tables provide an interface between the internal reality and the user's conventions for numbering the variables.

Because of the special meaning of *taut* functions, we don't "swap" them. Indeed, the former function *v→taut* might well have disappeared, if it was hidden; and if it was remotely accessible, it doesn't have the proper meaning for the new *u→taut*, because it is false when x_u is true. Instead, we compute the new *u→taut* from the new *v→taut*, which is identical to the former *u→taut*. (Think about it.)

⟨ Swap names and projection functions 142 ⟩ ≡

```
oo, j = u→name, k = v→name;
oooo, u→name = k, v→name = j, varmap[j] = v − varhead, varmap[k] = u − varhead;
oo, j = u→aux, k = v→aux;
if (j * k < 0) oo, u→aux = −j, v→aux = −k;    /* sign of aux stays with name */
o, j = u→proj, k = u→elt;
oo, u→proj = v→proj, u→elt = v→elt;
o, v→proj = j, v→elt = k;
o, v→taut = addr_(node_(u→taut)→lo);
```

This code is used in section 131.

143. The *swap* subroutine is now complete. I can safely declare it, since its sub-subroutines have already been declared.

⟨ Subroutines 8 ⟩ +≡
⟨ Declare the *swap* subroutine 131 ⟩

144. ⟨ Bubble sort to reestablish the natural variable order 144 ⟩ ≡

```
if (totalvars) {
  reorder_init();    /* prepare for reordering */
  for (v = firstvar→down; v; ) {
    if (oo, v→name > v→up→name) v = v→down;
    else {
      swap(v→up, v);
      if (v→up→up) v = v→up;
      else v = v→down;
    }
  }
  reorder_fin();    /* go back to normal processing */
}
```

This code is used in section 112.

145. Now we come to the *sift* routine, which finds the best position for a given variable when the relative positions of the others are left unchanged.

⟨ Sift on variable x_k 145 ⟩ ≡

```
{
  getkv; v = &varhead[varmap[k]];
  reorder_init();    /* prepare for reordering */
  sift(v);
  reorder_fin();    /* go back to normal processing */
}
```

This code is used in section 112.

146. At this point $v\to aux$ is the position of v among all active variables. Thus $v\to aux = 1$ if and only if $v\to up = \Lambda$ if and only if $v = firstvar$; $v\to aux = totalvars$ if and only if $v\to down = \Lambda$.

⟨ Subroutines 8 ⟩ +≡

```
void sift(var *v)
{
  register int pass, bestscore, origscore, swaps;
  var *u = v;
  double worstratio, saferatio;
  unsigned long long oldmems = mems, oldrmems = rmems, oldzmems = zmems;

  bestscore = origscore = totalnodes;
  worstratio = saferatio = 1.0;
  swaps = pass = 0;    /* first we go up or down; then we go down or up */
  if (o, totalvars − v→aux < v→aux) goto siftdown;
siftup: ⟨ Explore in the upward direction 147 ⟩;
siftdown: ⟨ Explore in the downward direction 148 ⟩;
wrapup: if (verbose & 4096)
    printf("sift␣x%d␣(%d->%d),␣%d␣saved,␣%.3f␣safe,␣%d␣swaps,␣(%llu,0,%llu)␣mems\n",
      u→name, v − varhead, u − varhead, origscore − bestscore, saferatio, swaps, mems − oldmems,
      zmems − oldzmems);
  oo, u→aux = −u→aux;    /* mark this level as having been sifted */
}
```

147. In a production version of this program, I would stop sifting in a given direction when the ratio *totalnodes*/*bestscore* exceeds some threshold. Here, on the other hand, I'm sifting completely; but I calculate the *saferatio* for which a production version would obtain results just as good as the complete sift.

⟨ Explore in the upward direction 147 ⟩ ≡

```
while (o, u→up) {
  swaps++, swap(u→up, u);
  u = u→up;
  if (bestscore > totalnodes) {    /* we've found an improvement */
    bestscore = totalnodes;
    if (saferatio < worstratio) saferatio = worstratio;
    worstratio = 1.0;
  } else if (totalnodes > worstratio * bestscore) worstratio = (double) totalnodes/bestscore;
}
if (pass ≡ 0) {    /* we want to go back to the starting point, then down */
  while (u ≠ v) {
    o, swaps++, swap(u, u→down);
    u = u→down;
  }
  pass = 1, worstratio = 1.0;
  goto siftdown;
}
while (totalnodes ≠ bestscore) {    /* we want to go back to an optimum level */
  swaps++, swap(u, u→down);
  u = u→down;
}
goto wrapup;
```

This code is used in section 146.

148. ⟨ Explore in the downward direction 148 ⟩ ≡

```
while (o, u→down) {
  swaps++, swap(u, u→down);
  u = u→down;
  if (bestscore > totalnodes) {    /* we've found an improvement */
    bestscore = totalnodes;
    if (saferatio < worstratio) saferatio = worstratio;
    worstratio = 1.0;
  } else if (totalnodes > worstratio * bestscore) worstratio = (double) totalnodes/bestscore;
}
if (pass ≡ 0) {    /* we want to go back to the starting point, then up */
  while (u ≠ v) {
    o, swaps++, swap(u→up, u);
    u = u→up;
  }
  pass = 1, worstratio = 1.0;
  goto siftup;
}
while (totalnodes ≠ bestscore) {    /* we want to go back to an optimum level */
  o, swaps++, swap(u→up, u);
  u = u→up;
}
goto wrapup;
```

This code is used in section 146.

149. The *siftall* subroutine sifts until every variable has found a local sweet spot. This is as good as it gets, unless the user elects to sift some more.

The order of sifting obviously affects the results. We could, for instance, sift first on a variable whose level has the most nodes. But Rudell tells me that nobody has found an ordering strategy that really stands out and outperforms the others. (He says, "It's a wash.") So I've adopted the first ordering that I thought of.

⟨ Subroutines 8 ⟩ +≡

```
void siftall(void)
{
  register var *v;
  reorder_init();
  for (v = firstvar; v; ) {
    if (o, v→aux < 0) {    /* we've already sifted this guy */
      v = v→down;
      continue;
    }
    sift(v);
  }
  reorder_fin();
}
```

150. Sifting is invoked automatically when the number of nodes is *toobig* or more. By default, the *toobig* threshold is essentially infinite, hence autosifting is disabled. But if a trigger of k is set, we'll set *toobig* to $k/100$ times the current size, and then to $k/100$ times the size after an autosift.

⟨ Reset the reorder trigger 150 ⟩ ≡

```
getk;
trigger = k/100.0;
if (trigger * totalnodes ≥ memsize) toobig = memsize;
else toobig = trigger * totalnodes;
```

This code is used in section 112.

151. ⟨ Invoke autosifting 151 ⟩ ≡

```
{
  if (verbose & (4096 + 8192))
    printf("autosifting␣(totalnodes=%d,␣trigger=%.2f,␣toobig=%d)\n", totalnodes, trigger, toobig);
  siftall();   /* hopefully totalnodes will decrease */
  if (trigger * totalnodes ≥ memsize) toobig = memsize;
  else toobig = trigger * totalnodes;
}
```

This code is used in section 109.

152. ⟨ Global variables 5 ⟩ +≡

```
double trigger;   /* multiplier that governs automatic sifting */
int toobig = memsize;   /* threshold for automatic sifting (initially disabled) */
```

153. Triage and housekeeping. Hmmm; we can't postpone the dirty work any longer. In emergency situations, garbage collection is a necessity. And occasionally, as a ZDD base grows, garbage collection is a nicety, to keep our house in order.

The *collect_garbage* routine frees up all of the nodes that are currently dead. Before it can do this, all references to those nodes must be eliminated, from the cache and from the unique tables. When the *level* parameter is nonzero, the cache is in fact entirely cleared.

```
⟨ Subroutines 8 ⟩ +≡
  void collect_garbage(int level)
  {
    register int k;
    var *v;
    node *p;
    last_ditch = 0;    /* see below */
    if (¬level) cache_purge();
    else {
      if (verbose & 512) printf("clearing␣the␣cache\n");
      for (k = 0; k < cachepages; k++) free_page(page_(cachepage[k]));
      cachepages = 0;
    }
    if (verbose & 512) printf("collecting␣garbage␣(%d/%d)\n", deadnodes, totalnodes);
    for (v = varhead; v < topofvars; v++) table_purge(v);
  }
```

154. The global variable *last_ditch* is set nonzero when we resort to garbage collection without a guarantee of gaining at least *totalnodes*/*deadfraction* free nodes in the process. If a last-ditch attempt fails, there's little likelihood that we'll get much further by eking out only a few more nodes each time; so we give up in that case.

155.
```
⟨ Global variables 5 ⟩ +≡
  int last_ditch;    /* are we backed up against the wall? */
```

156.
```
⟨ Subroutines 8 ⟩ +≡
  void attempt_repairs(void)
  {
    register int j, k;
    if (last_ditch) {
      printf("sorry␣---␣there's␣not␣enough␣memory;␣we␣have␣to␣quit!\n");
      ⟨ Print statistics about this run 7 ⟩;
      exit(−99);    /* we're outta here */
    }
    if (verbose & 512) printf("(making␣a␣last␣ditch␣attempt␣for␣space)\n");
    collect_garbage(1);    /* grab all the remaining space */
    cache_init();    /* initialize a bare-bones cache */
    last_ditch = 1;    /* and try one last(?) time */
  }
```

157. Mathematica output. An afterthought: It's easy to output a (possibly huge) file from which Mathematica will compute the generating function. (In fact, with ZDDs it's even easier than it was before.)

```
⟨ Print a Mathematica program for a generating function 157 ⟩ ≡
  getkf;
  math_print(f[k]);
  fprintf(stderr, "(generating␣function␣for␣f%d␣written␣to␣%s)\n", k, buf);
```

This code is used in section 112.

158. ⟨ Global variables 5 ⟩ +≡

```
  FILE *outfile;
  int outcount;      /* the number of files output so far */
```

159. ⟨ Subroutines 8 ⟩ +≡

```
  void math_print(node *p)
  {
    var *v;
    int k, s;
    node *q, *r;
    if (¬p) return;
    outcount++;
    sprintf(buf, "/tmp/bdd15-out%d.m", outcount);
    outfile = fopen(buf, "w");
    if (¬outfile) {
      fprintf(stderr, "I␣can't␣open␣file␣%s␣for␣writing!\n", buf);
      exit(−71);
    }
    fprintf(outfile, "g0=0\ng1=1\n");
    if (p > topsink) {
      mark(p);
      for (s = 0, v = topofvars − 1; v ≥ varhead; v−−)
        ⟨ Generate Mathematica outputs for variable v 160 ⟩;
      unmark(p);
    }
    fprintf(outfile, "g%x\n", id(p));
    fclose(outfile);
  }
```

160. ⟨ Generate Mathematica outputs for variable v 160 ⟩ ≡

```
  {
    for (k = 0; k < v→mask; k += sizeof(addr)) {
      q = fetchnode(v, k);
      if (q ∧ (q→xref + 1) < 0) {
        ⟨ Generate a Mathematica line for node q 161 ⟩;
      }
    }
  }
```

This code is used in section 159.

161. ⟨ Generate a Mathematica line for node q 161 ⟩ ≡

```
fprintf(outfile, "g%x=Expand[", id(q));
r = node_(q→lo);
fprintf(outfile, "g%x+z*", id(r));
r = node_(q→hi);
fprintf(outfile, "g%x]\n", id(r));
```

This code is used in section 160.

162. Index.

BDD15