November 24, 2020 at 13:23

1. Intro. This program is the fourteenth in a series of exploratory studies by which I'm attempting to gain first-hand experience with OBDD structures, as I prepare Section 7.1.4 of *The Art of Computer Programming*. It's basically the same as BDD11, but extended to include some rudimentary methods for changing and "sifting" the order of variables.

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

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

⟨const⟩ ← `c0` | `c1`
⟨var⟩ ← `x`⟨number⟩
⟨func⟩ ← `f`⟨number⟩
⟨atom⟩ ← ⟨const⟩ | ⟨var⟩ | ⟨func⟩
⟨expr⟩ ← ⟨unop⟩⟨atom⟩ | ⟨atom⟩⟨binop⟩⟨atom⟩ | ⟨atom⟩`[y]` |
⟨atom⟩⟨ternop⟩⟨atom⟩⟨ternop⟩⟨atom⟩
⟨command⟩ ← ⟨special⟩ | ⟨func⟩`=`⟨expr⟩ | ⟨func⟩`=.` | `y`⟨number⟩`=`⟨atom⟩ | `y`⟨number⟩`=.`

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

```
f1=x1^x2
f2=x3|x4
f1=f1&f2
f2=~f1
```

the function f_1 will be $(x_1 \oplus x_2) \wedge (x_3 \vee x_4)$, and f_2 will be $\neg f_1$. Then '`f1=.`' will undefine f_1.

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.

For simplicity, I do my own memory allocation in a big array called *mem*. The bottom part of that array is devoted to BDD 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 $mem[b[k \gg 12] + (k \mathbin{\&} {}^\#\texttt{fff})]$, for some array b of base addresses.

Each node of the BDD 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.

2. 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 10 ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Templates for subroutines 25 ⟩
  ⟨ Subroutines 7 ⟩
  main(int argc, char *argv[])
  {
    ⟨ Local variables 19 ⟩;
    ⟨ Check the command line 3 ⟩;
    ⟨ Initialize everything 5 ⟩;
    while (1) ⟨ Read a command and obey it; goto alldone if done 110 ⟩;
  alldone: ⟨ Print statistics about this run 6 ⟩;
    exit(0);    /* normal termination */
  }
```

3. **#define** *file_given* $(argc \equiv 2)$

```
⟨ Check the command line 3 ⟩ ≡
  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 2.

4. ⟨ Global variables 4 ⟩ ≡

```
  FILE *infile;    /* input file containing commands */
  int verbose = −1;    /* master control for debugging output; −1 gives all */
```

See also sections 8, 13, 22, 30, 40, 42, 51, 59, 101, 112, 132, 136, 154, 158, and 161.

This code is used in section 2.

5. ⟨ Initialize everything 5 ⟩ ≡

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

See also sections 9, 12, 23, 44, and 62.

This code is used in section 2.

6. 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
```

```
⟨ Print statistics about this run 6 ⟩ ≡
  printf("Job␣stats:\n");
  printf("␣␣%llu␣mems␣plus␣%llu␣rmems␣plus␣%llu␣zmems\n", mems, rmems, zmems);
  ⟨ Print total memory usage 18 ⟩;
```

This code is used in sections 2 and 159.

7. ⟨ Subroutines 7 ⟩ ≡

```
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\n", mems, rmems, zmems);
}
```

See also sections 14, 15, 16, 24, 26, 35, 41, 43, 45, 46, 49, 52, 53, 54, 55, 56, 58, 67, 71, 72, 73, 75, 77, 79, 82, 84, 86, 88, 91, 93, 95, 97, 104, 106, 108, 109, 131, 137, 138, 141, 145, 148, 151, 156, 159, and 162.

This code is used in section 2.

8. 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 4 ⟩ +≡
  unsigned long long mems, rmems, zmems;    /* mem counters */
```

9. ⟨ Initialize everything 5 ⟩ +≡

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

10. 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 BDD 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 10 ⟩ ≡
typedef unsigned int addr;

See also sections 11, 20, and 39.

This code is used in section 2.

11. Dynamic arrays. Before I get into the BDD 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 BDD 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 10⟩ +≡

```
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;
```

12. 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 5⟩ +≡

```
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;
oooo, botsink→index = gb_next_rand();
oooo, topsink→index = gb_next_rand();
totalnodes = 2;
nodeptr = topsink + 1;
pageptr = topofmem;
```

13. ⟨ Global variables 4 ⟩ +≡

```
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 = 10;
```

14. 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 7 ⟩ +≡

```
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;
}
```

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

⟨ Subroutines 7 ⟩ +≡

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

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

```
⟨ Subroutines 7 ⟩ +≡
  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;
  }
```

17. ⟨ If there are at least three free pages and at least three free nodes, **break** 17 ⟩ ≡

```
  j = (node *)(pageptr − 3) − nodeptr;
  if (j ≥ 0) {
    for (p = nodeavail; p ∧ j < 3; o, p = (node_(p→xref))) j++;
    if (j ≥ 3) break;
  }
```

This code is used in section 111.

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 6.

19. ⟨ Local variables 19 ⟩ ≡

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

See also section 115.

This code is used in section 2.

20. Variables and hash tables. Our BDD base represents functions on the variables x_v for $0 \le v < varsize$, 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 BDD 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 BDDs. If, for example, $logmaxhashsize = 19$ and $logpagesize = 12$, 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 10 ⟩ +≡

```
typedef struct var_struct {
  addr proj;   /* address of the projection function x_v */
  addr repl;   /* address of the replacement function y_v */
  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 */
  unsigned int timestamp;   /* time stamp for composition */
  int aux;   /* flag used by math_print or the sifting algorithm */
  struct var_struct *up, *down;   /* the neighboring active variables */
} var;
```

21. Every node p that branches on x_v in the BDD has a field *p→index*, whose leftmost *logvarsize* bits contain the index v. The rightmost $32 - logvarsize$ 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 $logvarsize + logmaxhashsize \le 32$.

```
#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[varsize]
```

⟨ Global variables 4 ⟩ +≡

```
var varhead[varsize];   /* basic info about each variable */
var *tvar = topofvars;   /* threshold for verbose printouts */
int varmap[varsize];   /* the variable that has a given name */
```

23. ⟨ Initialize everything 5 ⟩ +≡

```
for (k = 0; k < varsize; k++) varmap[k] = k;
```

24. The simplest nonconstant Boolean expression is a projection function, x_v. We access it with the following subroutine, creating it from scratch if necessary.

(The calling routine will have ensured that at least one free page and at least one free node exist when *projection* is invoked.)

Beware: Garbage collection might occur when *unique_find* is called here.

⟨ Subroutines 7 ⟩ +≡

```
node *projection(int v)
{
  register node *p;
  register var *hv = &varhead[v];
  o, p = node_(hv→proj);
  if (p) return p;    /* the projection function has already been created */
  o, hv→base[0] = addr_(reserve_page());    /* it won't be Λ */
  ⟨ Create a unique table for variable hv with size 2 28 ⟩;
  p = unique_find(&varhead[v], botsink, topsink);    /* see below */
  oooo, botsink→xref++, topsink→xref++;
  o, hv→proj = addr_(p);    /* p won't be Λ either */
  if (verbose & 2) printf("␣%x=x%d\n", id(p), v);
  o, hv→name = v;
  return p;
}
```

25. I sometimes like to use a subroutine before I'm in the mood to write its innards. In such cases, a pre-specification like the one given here allows me to procrastinate.

⟨ Templates for subroutines 25 ⟩ ≡

```
node *unique_find(var *v, node *l, node *h);
```

See also sections 27, 90, and 107.

This code is used in section 2.

26. Now, however, I'm ready to tackle the subroutine just named, *unique_find*, 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 BDD 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

$$(l\text{→}index \ll 3) \oplus (h\text{→}index \ll 2).$$

(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 $l = h$. 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 7 ⟩ +≡

```
  node *unique_find(var *v, node *l, node *h)
  {
    register int j, k, mask, free;
    register addr *hash;
    register node *p, *r;
    if (l ≡ h) {    /* easy case */
      return oo, l→xref−−, l;    /* l→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 29 ⟩;
    ⟨ Create a new node and return it 31 ⟩;
  }
```

27. ⟨ Templates for subroutines 25 ⟩ +≡

```
void recursively_revive(node *p);   /* recursive resuscitation */
void recursively_kill(node *p);   /* recursive euthanization */
void collect_garbage(int level);   /* invocation of the recycler */
```

28. 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) $= 0_{\rm LL}$;

⟨ Create a unique table for variable *hv* with size 2 28 ⟩ ≡

```
o, hv→free = 2, hv→mask = 7;
storenulls(hv→base[0]);   /* both slots start out Λ */
zmems++;
```

This code is used in section 24.

29. 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 29 ⟩ ≡

```
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 26.

30. ⟨ Global variables 4 ⟩ +≡

```
unsigned long long timer;
```

31. Brand-new nodes enter the fray here.

⟨ Create a new node and return it 31 ⟩ ≡

```
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 32 ⟩;
}
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 26.

32. 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* 32 ⟩ ≡

```
{
  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 33 ⟩
  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 34 ⟩;
  v→mask = newmask;   /* mems are counted after restarting */
  v→free = free + 1 + (newmask − mask)/sizeof(addr);
  goto restart;
}
```

This code is used in sections 31, 138, and 142.

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

⟨ Reserve new all-Λ pages for the bigger table 33 ⟩ ≡

```
{
  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 32.

34. Some subtle cases can arise at this point. For example, consider the hash table (a, Λ, Λ, b), with $\mathrm{hash}(a) = 3$ and $\mathrm{hash}(b) = 7$; 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 34 ⟩ ≡
  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 32.

35. 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 7 ⟩ +≡
  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;
    if (o, v→proj) {    /* v→proj ≠ 0 if and only if x_v exists */
      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 36 ⟩;
        }
      }
      deadnodes −= oldtotal − totalnodes, free += oldtotal − totalnodes;
      ⟨ Downsize the table if only a few entries are left 37 ⟩;
      o, v→free = free;
    }
  }
```

36. 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 36 ⟩ ≡

```
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 35.

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

⟨ Downsize the table if only a few entries are left 37 ⟩ ≡

```
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 38 ⟩;
  for (k = mask ≫ logpagesize; k > newmask ≫ logpagesize; k−−) o, free_page(page_(v→base[k]));
  v→mask = newmask;
}
```

This code is used in section 35.

38. 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 38 ⟩ ≡

```
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 37.

39. The cache. The other principal data structure we need, besides the BDD 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 BDD for which we've already computed $h = f \land g$, the cache should contain the information that $f \land g$ is known to be node h.

But that description is only approximately correct, because the cost of forgetting the value of $f \land g$ 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 $f \land g$ 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 BDD node, the result of an operation encoded by f, g, and h. This (f, g, h) encoding has several variants:

- If h is 0, then g is a "time stamp," and f points to a BDD node. This case is used for functional composition, when we want to invalidate a block of cache entries quickly by simply changing an external time stamp; items with a stale time stamp won't match any further cache lookups.
- If $0 < h \le$ *maxbinop*, then h denotes a binary operation on the BDD nodes f and g. For example, $h = 1$ stands for $\land$. The binary operations currently implemented are: and (1), but-not (2), not-but (4), xor (6), or (7), constrain (8), all-quantifier (9), no-quantifier (10), yes-quantifier (12), diff-quantifier (14), exists-quantifier (15).
- Otherwise (f, g, h) encodes a ternary operation on the three BDD nodes f, g, $h\ \&\ {-16}$. The four least-significant bits of h are used to identify the ternary operation involved: if-then-else (0), median (1), and-and (2), and-exist (3), not-yet-implemented (4–15).

```
#define memo_(a) ((memo *)(size_t)(a))
⟨Type definitions 10⟩ +≡
  typedef struct memo_struct {
    addr f;   /* first operand */
    addr g;   /* second operand, or time stamp */
    addr h;   /* third operand and/or operation code */
    addr r;   /* result */
  } memo;
```

40. 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 BDD. 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 4⟩ +≡
  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 */
```

41. 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 7 ⟩ +≡
  void print_memo(memo *m)
  {
    printf("%x", id(m→f));
    if (m→h ≡ 0) printf("[%d]", m→g);   /* time stamp */
    else 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);
  }
```

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

```
⟨ Global variables 4 ⟩ +≡
  char *binopname[] = {"","&",">","!","<","@","^","|","_","A","N","#","Y","$","D","E"};
  char *ternopname1[] = {"?",".","&","&","@","#","$","%","*","<","-","+","|","/","\\","~"};
  char *ternopname2[] = {":",".","&","E","@","#","$","%","*","<","-","+","|","/","\\","~"};
```

43. 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 7 ⟩ +≡

```
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;
}
```

44. ⟨ Initialize everything 5 ⟩ +≡

```
cache_init();
```

45. 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) $((f)\to index \ll 4) \oplus (((h)\ ?\ (g)\to index : addr\_(g)) \ll 5) \oplus (addr\_(h) \ll 6)$
#define *thememo*(s) $memo\_(cachepage[((s)\ \&\ cachemask) \gg logpagesize] + ((s)\ \&\ pagemask))$

⟨ Subroutines 7 ⟩ +≡

```
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;
}
```

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

⟨ Subroutines 7 ⟩ +≡

```
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 47 ⟩;
  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);
  }
}
```

47. ⟨ Double the cache size 47 ⟩ ≡

```
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 48 ⟩;
}
done:
```

This code is used in section 46.

48. ⟨ Recache the items in the bottom half 48 ⟩ ≡

```
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 47.

49. Before we purge elements from the unique tables, we need to purge all references to dead nodes from the cache. At the same time, we might as well purge items whose time stamp has expired.

```
⟨Subroutines 7⟩ +≡
  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 (m→h ≡ 0) {
            if (m→g ≠ thevar(node_(m→f))→timestamp) goto purge;
          } else {
            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 50⟩;
    cacheinserts = items;
  }
```

50. ⟨Downsize the cache if it has now become too sparse 50⟩ ≡

```
  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 49.

51. BDD structure. The reader of this program ought to be familiar with the basics of BDDs, namely the facts that a BDD 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 $l \neq h$ 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).

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

```
#define extsize 1000
```

```
⟨Global variables 4⟩ +≡
  node *f[extsize];   /* external pointers to functions in the BDD base */
```

52. 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 7⟩ +≡
  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 */
    }
  }
```

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

```
⟨Subroutines 7⟩ +≡
  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 */
    }
  }
```

54. Here's a simple routine that prints out the current BDDs, 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))
#define print_node_unmapped(p)
          printf("%x:␣(~%d?%x:%x)", id(p), thevar(p) − varhead, id((p)→lo), id((p)→hi))
```

⟨ Subroutines 7 ⟩ +≡

```
void print_base(int marked, int unmapped)
{
  register int j, k;
  register node *p;
  register var *v;

  for (v = varhead; v < topofvars; v++)
    if (v→proj) {
      for (k = 0; k < v→mask; k += sizeof(addr)) {
        p = fetchnode(v, k);
        if (p ∧ (¬marked ∨ (p→xref + 1) < 0)) {
          if (unmapped) print_node_unmapped(p); else print_node(p);
          if (marked ∨ p→xref ≡ 0) printf("\n");
          else printf("␣(%d)\n", p→xref);
        }
      }
      if (¬marked ∧ v→repl) printf("y%d=%x\n", v→name, id(v→repl));
    }
  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]));
  }
}
```

55. The marking feature is useful when we want to print out only a single BDD.

⟨ Subroutines 7 ⟩ +≡

```
void print_function(node *p, int unmapped)
{
  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, unmapped);
    unmark(p);
  }
  mems = savemems, rmems = savermems;
}
```

56. ⟨ Subroutines 7 ⟩ +≡

```
void print_profile(node *p)
{
  unsigned long long savemems = mems, savermems = rmems;
  register int j, k, tot;
  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++)
      if (v→proj) {
        ⟨ Print the number of marked nodes that branch on v 57 ⟩;
      }
    unmark(p);
    printf(" 2 (total %d)\n", tot + 2); /* the sinks */
  }
  mems = savemems, rmems = savermems;
}
```

57. ⟨ Print the number of marked nodes that branch on v 57 ⟩ ≡

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

This code is used in section 56.

58. In order to deal efficiently with large BDDs, we've introduced highly redundant data structures, including things like hash tables and the cache. Furthermore, we assume that every BDD node p has a redundant field $p\to xref$, which counts the total number of branch nodes, external nodes, replacement functions, 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 7 ⟩ +≡
#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 60 ⟩;
    ⟨ Check the reference counts 66 ⟩;
    ⟨ Check the unique tables 68 ⟩;
    ⟨ Check the cache 69 ⟩;
    ⟨ Check the list of free pages 70 ⟩;
    mems = savemems;
  }
#endif
```

59. 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, projection functions, and replacement functions.

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

⟨ Global variables 4 ⟩ +≡
#if includesanity
  char smem[memsize];    /* the shadow memory */
#endif
```

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

⟨ Build the shadow memory 60 ⟩ ≡
  for $(p = botsink;\ p < nodeptr;\ p{+}{+})$ $ghost(p)\to xref = 0, ghost(p)\to index = -1;$
  ⟨ Check the list of free nodes 64 ⟩;
  ⟨ Compute the ghost index fields 65 ⟩;
  for $(count = 2, p = topsink + 1;\ p < nodeptr;\ p{+}{+})$
    if $(ghost(p)\to xref \ne -1)$ {
      $count{+}{+};$
      if $(\neg legit(node\_(p\to lo)) \lor \neg legit(node\_(p\to hi)))$ *complain*("bad␣pointer")
      else if $(node\_(thevar(p)\to proj) \equiv \Lambda)$ *complain*("bad␣var")
      else if $(p\to lo \equiv p\to hi)$ *complain*("lo=hi")
      else {
        ⟨ Check that p is findable in the unique table 63 ⟩;
        if $(node\_(p\to lo) > topsink \land thevar(p) \ge thevar(node\_(p\to lo)))$ *complain*("bad␣lo␣rank");
        if $(node\_(p\to hi) > topsink \land thevar(p) \ge thevar(node\_(p\to hi)))$ *complain*("bad␣hi␣rank");
        if $(p\to xref \ge 0)$ { /* dead nodes don't point */
          $q = ghost(p);$
          $q\to lo = ghost(p\to lo)\to xref, ghost(p\to lo)\to xref = addr\_(\&(p\to lo));$
          $q\to hi = ghost(p\to hi)\to xref, ghost(p\to hi)\to xref = addr\_(\&(p\to hi));$
        }
      }
    }
  if $(count \ne totalnodes)$ *printf* ("!␣totalnodes␣should␣be␣%d,␣not␣%d\n", *count*, *totalnodes*);
  if $(extra \ne totalnodes)$ *printf* ("!␣%d␣nodes␣have␣leaked\n", $extra - totalnodes$);

This code is used in section 58.

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

62. ⟨ Initialize everything 5 ⟩ +≡
  if (**sizeof**(**node**) ≠ 16) {
    fprintf (*stderr*, "Sorry,␣I␣assume␣that␣sizeof(node)␣is␣16!\n");
    exit(−3);
  }

63. ⟨ Check that p is findable in the unique table 63 ⟩ ≡

```
{
  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 60.

64. ⟨ Check the list of free nodes 64 ⟩ ≡

```
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 60.

65. ⟨ Compute the ghost index fields 65 ⟩ ≡

```
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 (v→repl) {
      if (¬legit(node_(v→repl)))
        printf("!␣illegal␣replacement␣function␣for␣level␣%d\n", v − varhead);
      else ghost(v→repl)→index++;
    }
  }
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 60.

66. ⟨ Check the reference counts 66 ⟩ ≡

```
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 58.

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

⟨ Subroutines 7 ⟩ +≡

```
#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
```

68. We've seen that every superlegitimate 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 68 ⟩ ≡
  extra = topofmem − pageptr;   /* this many pages allocated */
  for (v = varhead; v < topofvars; v++)
    if (v→proj) {
      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 58.

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

```
⟨ Check the cache 69 ⟩ ≡
  {
    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 (m→h > 0) {
            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 58.

70. 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 70 ⟩ ≡

```
{
  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 58.

71. 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 7 ⟩ +≡

```
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++;
}
```

72. 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 7 ⟩ +≡

```
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 −−;
}
```

73. Binary operations. OK, now we've got a bunch of powerful routines for making and maintaining BDDs, and it's time to have fun. Let's start with a typical synthesis routine, which constructs the BDD for $f \wedge g$ from the BDDs 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 7⟩ +≡
```
node *and_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 ≤ topsink) {
    if (f ≡ topsink) return oo, g→xref++, g;    /* 1 ∧ g = g */
    return oo, botsink→xref++, botsink;    /* 0 ∧ g = 0 */
  }
  oo, r = cache_lookup(f, g, node_(1));    /* two mems for f→index, g→index */
  if (r) return r;
  ⟨Find f ∧ g recursively 74⟩;
}
```

74. 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 74⟩ ≡
```
rmems++;    /* track recursion overhead */
vf = thevar(f);    /* f→index has already been fetched */
vg = thevar(g);    /* g→index has already been fetched */
if (vf < vg) v = vf;
else v = vg;    /* choose the top variable, v */
r0 = and_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g));
if (¬r0) return Λ;    /* oops, trouble */
r1 = and_rec((vf ≡ v ? node_(f→hi) : f), (vg ≡ v ? node_(g→hi) : g));
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_(1), r);
}
return r;
```
This code is used in section 73.

75. Of course $f \lor g$ is dual to $f \land g$. I could have combined the two routines into one, but what the heck; a long program is more impressive.

⟨Subroutines 7⟩ +≡
```
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 ≤ topsink) {
    if (f ≡ topsink) return oo, topsink→xref++, topsink;   /* 1 ∨ g = 1 */
    return oo, g→xref++, g;   /* 0 ∨ g = g */
  }
  oo, r = cache_lookup(f, g, node_(7));
  if (r) return r;
  ⟨Find f ∨ g recursively 76⟩;
}
```

76. ⟨Find $f \lor g$ recursively 76⟩ ≡
```
rmems++;   /* track recursion overhead */
vf = thevar(f);
vg = thevar(g);
if (vf < vg) v = vf;
else v = vg;   /* choose the top variable, v */
r0 = or_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g));
if (¬r0) return Λ;   /* oops, trouble */
r1 = or_rec((vf ≡ v ? node_(f→hi) : f), (vg ≡ v ? node_(g→hi) : g));
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 75.

77. Exclusive or is much the same.

⟨Subroutines 7⟩ +≡

```
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 78⟩;
}
```

78. After discovering that $f \oplus g = r$, we could also put $f \oplus r = g$ and $g \oplus r = f$ into the cache. I tried that, in the first draft of this code. Unfortunately it cost more than it saved.

⟨Find $f \oplus g$ recursively 78⟩ ≡

```
rmems++;   /* track recursion overhead */
if (f ≡ topsink) vf = 0, v = vg = thevar(g);
else {
  vf = thevar(f), vg = thevar(g);
  if (vf < vg) v = vf;
  else v = vg;   /* choose the top variable, v */
}
r0 = xor_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g));
if (¬r0) return Λ;   /* oops, trouble */
r1 = xor_rec((vf ≡ v ? node_(f→hi) : f), (vg ≡ v ? node_(g→hi) : g));
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 77.

79. Now for variety, let's do the "constrain" operator of Coudert and Madre. The function $f \downarrow g$ returned by this routine depends on the current variable ordering. In brief, $f \downarrow g = 0$ if g is identically zero; otherwise $(f \downarrow g)(x) = f(y)$ when y is the first element of the sequence x, $x \oplus 1$, $x \oplus 2$, ... such that $g(y) = 1$. (This definition treats $x = (x_0 \ldots x_n)_2$ and $y = (y_0 \ldots y_n)_2$ as binary numbers.)

⟨Subroutines 7⟩ +≡

```
node *constrain_rec(node *f, node *g)
{
  var *v, *vf, *vg;
  node *r, *r0, *r1;
  if (g ≡ botsink) return oo, botsink→xref++, botsink;     /* f ↓ 0 = 0 */
  if (g ≡ topsink ∨ f ≤ topsink) return oo, f→xref++, f;
      /* f ↓ 1 = f, 0 ↓ g = 0; also 1 ↓ g = 1 if g ≠ 0 */
  if (f ≡ g) return oo, topsink→xref++, topsink;     /* f ↓ f = 1 */
  oo, r = cache_lookup(f, g, node_(8));
  if (r) return r;
  ⟨Find f ↓ g recursively 80⟩;
}
```

80. ⟨Find $f \downarrow g$ recursively 80⟩ ≡

```
  rmems++;     /* track recursion overhead */
  vf = thevar(f);
  vg = thevar(g);
  if (vf < vg) v = vf;
  else {
    v = vg;     /* choose the top variable, v */
    if (o, node_(g→lo) ≡ botsink) {
      oo, r = constrain_rec((vf ≡ v ? o, node_(f→hi) : f), node_(g→hi));
      goto shortcut;
    }
    if (o, node_(g→hi) ≡ botsink) {
      oo, r = constrain_rec((vf ≡ v ? o, node_(f→lo) : f), node_(g→lo));
      goto shortcut;
    }
  }
  r0 = constrain_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g));
  if (¬r0) return Λ;     /* oops, trouble */
  r1 = constrain_rec((vf ≡ v ? node_(f→hi) : f), (vg ≡ v ? node_(g→hi) : g));
  if (¬r1) {
    deref(r0);     /* too bad, but we have to abort in midstream */
    return Λ;
  }
  r = unique_find(v, r0, r1);
shortcut: 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 79.

81. Quantifiers. Are we having fun yet? Sure, and quantifiers are even funner.

In each of the following routines, the second operand g is supposed to be a conjunction of positive literals, like $x_2 \wedge x_4 \wedge x_5$. Then, for example, $f \mathbin{\mathrm{A}} g$ stands for $\forall x_2 \forall x_4 \forall x_5 \, f(x_1, \ldots, x_n)$.

The program doesn't actually bother to check that g has this form. But if g is a general function, the meaning is dependent on the ordering of variables; consider, for instance, the case $g = \bar{x}_1 \vee x_2$. The user who tries general functions had better be aware (or beware) of this fact.

If g is a conjunction of k literals, and if they all have the highest possible rank (so that they occur at the bottom of the BDDs), quantification takes linear time in the size of the BDD for f. But if the literals have low rank and occur near the top, the running time can be as bad as the 2^kth power of that BDD size(!).

82. The first case, existential quantification ($f \mathbin{\mathrm{E}} g$), is typical. Notice that we don't use the cache unless there are multiple references to f. The number of references to g is immaterial here, since we're treating g as a one-dimensional list of literals.

```
⟨Subroutines 7⟩ +≡
  node *exist_rec(node *f, node *g)
  {
    var *v, *vg;
    node *r, *r0, *r1;
  restart: if (g ≤ topsink) return oo, f→xref++, f;    /* f E 1 = f */
    if (f ≤ topsink) return oo, f→xref++, f;    /* 0 E g = 0, 1 E g = 1 */
    v = thevar(f);
    vg = thevar(g);
    if (v > vg) {
      o, g = node_(g→hi); goto restart;    /* f doesn't depend on vg */
    }
    oo, r = cache_lookup(f, g, node_(15));
    if (r) return r;
    ⟨Find f E g recursively 83⟩;
  }
```

83. When the top variable of g is the same as the top variable of f, we always descend to the *hi* branch below node g, since g is supposed to be a conjunction.

⟨ Find f E g recursively 83 ⟩ ≡

```
rmems++;   /* track recursion overhead */
o, r0 = exist_rec(node_(f→lo), (vg ≡ v ? o, node_(g→hi) : g));
if (¬r0) return Λ;   /* oops, trouble */
if (r0 ≡ topsink ∧ vg ≡ v) {
  r = r0;
  goto gotr;
}
r1 = exist_rec(node_(f→hi), (vg ≡ v ? node_(g→hi) : g));
if (¬r1) {
  deref(r0);   /* too bad, but we have to abort in midstream */
  return Λ;
}
if (vg > v) r = unique_find(v, r0, r1);
else {
  r = or_rec(r0, r1);   /* existential quantification happens here */
  deref(r0); deref(r1);   /* we're done with r0 and r1 */
}
gotr: if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%xE%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(15), r);
}
return r;
```

This code is used in section 82.

84. The code for universal quantification (∀) is almost line-for-line identical to the code for existential quantification.

⟨ Subroutines 7 ⟩ +≡

```
node *all_rec(node *f, node *g)
{
  var *v, *vg;
  node *r, *r0, *r1;
restart: if (g ≤ topsink) return oo, f→xref++, f;   /* f A 1 = f */
  if (f ≤ topsink) return oo, f→xref++, f;   /* 0 A g = 0, 1 A g = 1 */
  v = thevar(f);
  vg = thevar(g);
  if (v > vg) {
    o, g = node_(g→hi); goto restart;   /* f doesn't depend on vg */
  }
  oo, r = cache_lookup(f, g, node_(9));
  if (r) return r;
  ⟨ Find f A g recursively 85 ⟩;
}
```

85. ⟨ Find f A g recursively 85 ⟩ ≡

```
rmems++;   /* track recursion overhead */
o, r0 = all_rec(node_(f→lo), (vg ≡ v ? o, node_(g→hi) : g));
if (¬r0) return Λ;   /* oops, trouble */
if (r0 ≡ botsink ∧ vg ≡ v) {
  r = r0;
  goto gotr;
}
r1 = all_rec(node_(f→hi), (vg ≡ v ? node_(g→hi) : g));
if (¬r1) {
  deref(r0);   /* too bad, but we have to abort in midstream */
  return Λ;
}
if (vg > v) r = unique_find(v, r0, r1);
else {
  r = and_rec(r0, r1);   /* universal quantification happens here */
  deref(r0); deref(r1);   /* we're done with r0 and r1 */
}
gotr: if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%xA%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(9), r);
}
return r;
```

This code is used in section 84.

86. The Boolean difference, f D g, is easier, because $f \oplus f = 0$.

⟨ Subroutines 7 ⟩ +≡

```
node *diff_rec(node *f, node *g)
{
  var *v, *vg;
  node *r, *r0, *r1;
  if (g ≤ topsink) return oo, f→xref++, f;   /* f D 1 = f */
  if (f ≤ topsink) return oo, botsink→xref++, botsink;   /* 0 D g = 1 D g = 0 when g isn't constant */
  v = thevar(f);
  vg = thevar(g);
  if (v > vg) return oo, botsink→xref++, botsink;   /* f doesn't depend on vg */
  oo, r = cache_lookup(f, g, node_(14));
  if (r) return r;
  ⟨ Find f D g recursively 87 ⟩;
}
```

87. ⟨Find f D g recursively 87⟩ ≡

```
rmems++;   /* track recursion overhead */
o, r0 = diff_rec(node_(f→lo), (vg ≡ v ? o, node_(g→hi) : g));
if (¬r0) return Λ;   /* oops, trouble */
r1 = diff_rec(node_(f→hi), (vg ≡ v ? node_(g→hi) : g));
if (¬r1) {
  deref(r0);   /* too bad, but we have to abort in midstream */
  return Λ;
}
if (vg > v) r = unique_find(v, r0, r1);
else {
  r = xor_rec(r0, r1);   /* differencing happens here */
  deref(r0); deref(r1);   /* we're done with r0 and r1 */
}
if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%xD%x␣(level␣%d)\n", id(r), id(f), id(g), v − varhead);
  cache_insert(f, g, node_(14), r);
}
return r;
```

This code is used in section 86.

88. The other two quantifiers are unusual; in fact, I haven't seen them in any books, although I haven't read all possible books. Still, the concept is natural enough, when g is a *single* variable x_v. In that case, f Y x_v is the function of the remaining variables such that $f(x_1, \ldots, x_n) = x_v$; and f N x_v is the similar function that makes $f(x_1, \ldots, x_n) = \bar{x}_v$.

For example, the function $f(x_1, \ldots, x_n)$ is monotonic if and only if f N $x_v = 0$ for $1 \le v \le n$.

On the other hand, these yes-no quantifiers make little sense when g involves more than one literal, because the result depends on the variable ordering. It's best to forget that general case—don't even *think* about it. Just enjoy the case that works.

⟨Subroutines 7⟩ +≡

```
node *yes_no_rec(int curop, node *f, node *g)
{
  var *v, *vg;
  node *r, *r0, *r1;
  if (g ≤ topsink) return oo, f→xref++, f;   /* f Y 1 = f N 1 = f */
  if (f ≤ topsink) return oo, botsink→xref++, botsink;
      /* 0 Y g = 1 Y g = 0 N g = 1 N g = 0 when g isn't constant */
  v = thevar(f);
  vg = thevar(g);
  if (v > vg) return oo, botsink→xref++, botsink;   /* f doesn't depend on vg */
  oo, r = cache_lookup(f, g, node_(curop));
  if (r) return r;
  ⟨Find f Y g or f N g recursively 89⟩;
}
```

89. ⟨ Find f Y g or f N g recursively 89 ⟩ ≡

```
rmems++;   /* track recursion overhead */
o, r0 = yes_no_rec(curop, node_(f→lo), (vg ≡ v ? o, node_(g→hi) : g));
if (¬r0) return Λ;   /* oops, trouble */
if (r0 ≤ topsink ∧ vg ≡ v) {
  if ((r0 ≡ topsink) ≡ (curop ≡ 12)) {
    r = botsink; deref(r0); botsink→xref++;
    goto gotr;
  }
}
r1 = yes_no_rec(curop, node_(f→hi), (vg ≡ v ? node_(g→hi) : g));
if (¬r1) {
  deref(r0);   /* too bad, but we have to abort in midstream */
  return Λ;
}
if (vg > v) r = unique_find(v, r0, r1);
else {
  if (curop ≡ 12) r = mux_rec(r0, botsink, r1);   /* f Y g = r̄0 ∧ r1 */
  else r = mux_rec(r1, botsink, r0);   /* f N g = r0 ∧ r̄1 */
  deref(r0); deref(r1);   /* we're done with r0 and r1 */
}
gotr: if (r) {
  if ((verbose & 128) ∧ (v < tvar))
    printf("␣␣␣%x=%x%s%x␣(level␣%d)\n", id(r), id(f), binopname[curop], id(g), v − varhead);
  cache_insert(f, g, node_(curop), r);
}
return r;
```

This code is used in section 88.

90. Stay tuned: The *mux_rec* subroutine is coming soon.

⟨ Templates for subroutines 25 ⟩ +≡

```
node *mux_rec(node *f, node *g, node *h);
```

91. 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.

Two special cases are worthy of note: $h = 1$ gives "f implies g"; $g = 0$ gives "not f but g." I could have implemented those cases as binary operators, but I chose to let them take the slightly slower ternary route. (I'm usually a fan of speed, but this program is already long enough.)

```
⟨Subroutines 7⟩ +≡
  node *mux_rec(node *f, node *g, node *h)
  {
    var *v, *vf, *vg, *vh;
    node *r, *r0, *r1;
    if (f ≤ topsink) {
      if (f ≡ topsink) return oo, g→xref++, g;     /* (1? g: h) = g */
      return oo, h→xref++, h;     /* (0? g: h) = h */
    }
    if (g ≡ f ∨ g ≡ topsink) return or_rec(f, h);     /* (f? f: h) = (f? 1: h) = f ∨ h */
    if (h ≡ f ∨ h ≡ botsink) return and_rec(f, g);     /* (f? g: f) = (f? g: 0) = f ∧ g */
    if (g ≡ h) return oo, g→xref++, g;     /* (f? g: g) = g */
    if (g ≡ botsink ∧ h ≡ topsink) return xor_rec(topsink, f);     /* (f? 0: 1) = 1 ⊕ f */
    oo, r = cache_lookup(f, g, h);
    if (r) return r;
    ⟨Find (f? g: h) recursively 92⟩;
  }
```

92. ⟨Find (f? g: h) recursively 92⟩ ≡

```
  rmems++;     /* track recursion overhead */
  v = vf = thevar(f);
  if (g ≡ botsink) vg = topofvars; else {
    vg = thevar(g); if (v > vg) v = vg;
  }
  if (h ≡ topsink) vh = topofvars; else {
    o, vh = thevar(h); if (v > vh) v = vh;
  }
  r0 = mux_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g), (vh ≡ v ? o, node_(h→lo) : h));
  if (¬r0) return Λ;     /* oops, trouble */
  r1 = mux_rec((vf ≡ v ? o, node_(f→hi) : f), (vg ≡ v ? o, node_(g→hi) : g), (vh ≡ v ? o, node_(h→hi) : h));
  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 91.

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

⟨ Subroutines 7 ⟩ +≡

```
node *med_rec(node *f, node *g, node *h)
{
  var *v, *vf, *vg, *vh;
  node *r, *r0, *r1;
  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 ≤ topsink) {
    if (f ≡ topsink) return or_rec(g, h);   /* ⟨1gh⟩ = g ∨ h */
    return and_rec(g, h);   /* ⟨0gh⟩ = g ∧ h */
  }
  if (f ≡ g) return oo, f→xref++, f;   /* ⟨ffh⟩ = f */
  if (g ≡ h) return oo, g→xref++, g;   /* ⟨fgg⟩ = g */
  oo, r = cache_lookup(f, g, node_(addr_(h) + 1));
  if (r) return r;
  ⟨ Find ⟨fgh⟩ recursively 94 ⟩;
}
```

94. ⟨ Find ⟨fgh⟩ recursively 94 ⟩ ≡

```
rmems++;   /* track recursion overhead */
vf = thevar(f);
vg = thevar(g);
o, vh = thevar(h);
if (vf < vg) v = vf; else v = vg;
if (v > vh) v = vh;   /* choose the top variable, v */
r0 = med_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g), (vh ≡ v ? o, node_(h→lo) : h));
if (¬r0) return Λ;   /* oops, trouble */
r1 = med_rec((vf ≡ v ? o, node_(f→hi) : f), (vg ≡ v ? o, node_(g→hi) : g), (vh ≡ v ? o, node_(h→hi) : h));
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, node_(addr_(h) + 1), r);
}
return r;
```

This code is used in section 93.

95. We can also exploit the symmetry of $f \wedge g \wedge h$. (If I had lots of time, I could play similarly with $f \vee g \vee h$ and $f \oplus g \oplus h$.)

⟨ Subroutines 7 ⟩ +≡

```
node *and_and_rec(node *f, node *g, node *h)
{
  var *v, *vf, *vg, *vh;
  node *r, *r0, *r1;
  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 ≤ topsink) {
    if (f ≡ topsink) return and_rec(g, h);    /* 1 ∧ g ∧ h = g ∧ h */
    return oo, botsink→xref++, botsink;    /* 0 ∧ g ∧ h = 0 */
  }
  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 */
  oo, r = cache_lookup(f, g, node_(addr_(h) + 2));
  if (r) return r;
  ⟨ Find f ∧ g ∧ h recursively 96 ⟩;
}
```

96. ⟨ Find $f \wedge g \wedge h$ recursively 96 ⟩ ≡

```
rmems++;    /* track recursion overhead */
vf = thevar(f);
vg = thevar(g);
o, vh = thevar(h);
if (vf < vg) v = vf; else v = vg;
if (v > vh) v = vh;    /* choose the top variable, v */
r0 = and_and_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g), (vh ≡ v ? o, node_(h→lo) : h));
if (¬r0) return Λ;    /* oops, trouble */
r1 = and_and_rec((vf ≡ v ? o, node_(f→hi) : f), (vg ≡ v ? o, node_(g→hi) : g), (vh ≡ v ? o, node_(h→hi) : h));
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, node_(addr_(h) + 2), r);
}
return r;
```

This code is used in section 95.

97. The final ternary operation computes $(f \wedge g)\,\mathrm{E}\,h$, when h is a conjunction of positive literals as before. Ken McMillan noticed that one can often save a lot of time computing this ternary function recursively, instead of first forming $f \wedge g$. We combine the ideas of *and_rec* and *exist_rec* here.

⟨Subroutines 7⟩ +≡

```
node *and_exist_rec(node *f, node *g, node *h)
{
    var *v, *vf, *vg, *vh;
    node *r, *r0, *r1;
restart: if (h ≤ topsink) return and_rec(f, g);    /* (f ∧ g) E 1 = f ∧ g */
    if (f ≡ g) return exist_rec(f, h);    /* (f ∧ f) E h = f E h */
    if (f > g) r = f, f = g, g = r;    /* wow */
    if (f ≤ topsink) {
        if (f ≡ topsink) return exist_rec(g, h);    /* (1 ∧ g) E h = g E h */
        return oo, botsink→xref++, botsink;    /* (0 ∧ d) E h = 0 */
    }
    oo, r = cache_lookup(f, g, node_(addr_(h) + 3));
    if (r) return r;
    vf = thevar(f);
    vg = thevar(g);
    o, vh = thevar(h);
    if (vf < vg) v = vf; else v = vg;
    if (v > vh) {
        o, h = node_(h→hi); goto restart;    /* f and g don't depend on vh */
    }
    ⟨Find (f ∧ g) E h recursively 98⟩;
}
```

98. ⟨Find $(f \wedge g)\,\mathrm{E}\,h$ recursively 98⟩ ≡

```
rmems++;    /* track recursion overhead */
r0 = and_exist_rec((vf ≡ v ? o, node_(f→lo) : f), (vg ≡ v ? o, node_(g→lo) : g), (vh ≡ v ? o, node_(h→hi) : h));
if (¬r0) return Λ;    /* oops, trouble */
if (r0 ≡ topsink ∧ vg ≡ v) {
    r = r0;
    goto gotr;
}
r1 = and_exist_rec((vf ≡ v ? node_(f→hi) : f), (vg ≡ v ? node_(g→hi) : g), (vh ≡ v ? node_(h→hi) : h));
if (¬r1) {
    deref(r0);    /* too bad, but we have to abort in midstream */
    return Λ;
}
if (vh > v) r = unique_find(v, r0, r1);
else {
    r = or_rec(r0, r1);    /* existential quantification happens here */
    deref(r0); deref(r1);    /* we're done with r0 and r1 */
}
gotr: if (r) {
    if ((verbose & 128) ∧ (v < tvar))
        printf("␣␣␣%x=%x&%xE%x␣(level␣%d)\n", id(r), id(f), id(g), id(h), v − varhead);
    cache_insert(f, g, node_(addr_(h) + 3), r);
}
return r;
```

This code is used in section 97.

99. Composition of functions. Now we're ready for the biggie, the most powerful recursive command in our arsenal: Given the BDDs for a master function $f(x_0,\ldots,x_n)$ and for replacement functions $y_0(x_0,\ldots,x_n)$, …, $y_n(x_0,\ldots,x_n)$, construct the BDD for the grand composition

$$F(x_0,\ldots,x_n)=f\bigl(y_0(x_0,\ldots,x_n),\ldots,y_n(x_0,\ldots,x_n)\bigr).$$

This daunting task can actually be accomplished with a surprisingly short recursive program.

Of course the result might be huge, so the running time can be humongous in bad cases. But in not-so-bad cases, the running time is not bad at all. Let's be optimistic.

The replacements y_k are specified by individual commands like '`y3=f7`' or '`y6=x5`' or '`y2=c1`'; see the syntax at the beginning of this program. One can also cancel a previous replacement by saying, for example, '`y3=.`'. If no replacement y_k is currently specified, the identity function $y_k=x_k$ is assumed.

100. Our main tool for efficiency in not-so-bad cases is the cache, so that previously known results needn't be recalculated. Each node in the BDD base represents a function of the variables $(x_v,x_{v+1},\ldots)$, for some variable index v, and we can let the cache remember if we've already composed this function with the replacements $(y_v,y_{v+1},\ldots)$. When the user changes y_k, all results in the cache for functions with $v\le k$ are potentially incorrect; but the results for functions with $v>k$ remain usable.

Therefore we maintain a time stamp for each variable x_k. The time stamps for x_0 through x_k are increased whenever y_k is updated, so that cache entries for the composition of such functions won't match when *cache_lookup* is called.

The time stamp for x_v is zero if and only if all of y_v, y_{v+1}, … are currently undefined.

Two global variables control this process: The variable *timestamp_changed* is either 0 or 1; the difference *timestamp* − *timestamp_changed* is the number of times we've performed a composition operation.

101. ⟨ Global variables 4 ⟩ +≡

```
int timestamp;   /* the number of distinct compositions done or prepared */
int timestamp_changed;   /* has it changed since the last composition was done? */
```

102. Here's how the individual time stamps are updated when the user gives a new specification for y_k:

⟨ Assign q as the new value of $v\to repl$ 102 ⟩ ≡

```
v→repl = addr_(q);
if (q ≡ Λ) {   /* cancel a previous replacement */
  if (v + 1 < topofvars ∧ (o, (v + 1)→timestamp ≠ 0)) goto newstamps;
  while (v ≥ varhead ∧ (o, v→repl ≡ 0)) o, v→timestamp = 0, v−−;
      /* cancel the previous timestamp */
  if (v ≥ varhead) goto newstamps;
  timestamp −= timestamp_changed, timestamp_changed = 0;   /* all clear */
} else {
  oo, q→xref ++;
newstamps: if (¬timestamp_changed) timestamp_changed = 1, timestamp ++;
      /* N.B.: timestamp won't become zero */
  while (v ≥ varhead ∧ (o, v→timestamp ≠ timestamp)) o, v→timestamp = timestamp, v−−;
      /* this is done even when v→proj = 0 */
}
```

This code is used in section 128.

103. A special adjustment to the time stamps is needed when we're wiping out the whole cache.

⟨ Clear out the time stamps 103 ⟩ ≡
```
if (varhead[0].timestamp ≡ 0) o, timestamp = timestamp_changed = 0;
else {
  timestamp = timestamp_changed = 1;
  for (v = varhead; v < topofvars ∧ (o, v→timestamp); v++) o, v→timestamp = 1;
}
```
This code is used in section 156.

104. Okay, we're ready for the master recursion.

⟨ Subroutines 7 ⟩ +≡
```
node *compose_rec(node *f)
{
  var *vf;
  node *r, *r0, *r1;
  if (f ≤ topsink) return oo, f→xref++, f;   /* f is constant */
  o, vf = thevar(f);
  if (o, vf→timestamp ≡ 0) return oo, f→xref++, f;   /* f doesn't depend on y's */
  o, r = cache_lookup(f, node_(vf→timestamp), node_(0));
  if (r) return r;
  ⟨ Find f(y_0, ...) recursively 105 ⟩;
}
```

105. The computations here look basically the same as those we've been seeing in previous recursions. But in fact there is a huge difference: The functions *r0* and *r1* now can involve *all* variables, not just the variables near the bottom of the BDDs.

⟨ Find $f(y_0, \ldots)$ recursively 105 ⟩ ≡
```
rmems++;   /* track recursion overhead */
o, r0 = compose_rec(node_(f→lo));
if (¬r0) return Λ;   /* oops, trouble */
r1 = compose_rec(node_(f→hi));
if (¬r1) {
  deref(r0);   /* too bad, but we have to abort in midstream */
  return Λ;
}
if (o, vf→repl) r = node_(vf→repl); else r = node_(vf→proj);
r = mux_rec(r, r1, r0);   /* replacement happens here */
deref(r0); deref(r1);   /* we're done with r0 and r1 */
if (r) {
  if ((verbose & 128) ∧ (vf < tvar))
    printf("␣␣␣%x=%x[%u]␣(level␣%d)\n", id(r), id(f), vf→timestamp, vf − varhead);
  cache_insert(f, node_(vf→timestamp), node_(0), r);
}
return r;
```
This code is used in section 104.

106. 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 7 ⟩ +≡

```
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 1: r = and_rec(f, g); break;   /* f ∧ g */
    case 2: r = mux_rec(g, botsink, f); break;   /* f ∧ ḡ = (g? 0: f) */
    case 4: r = mux_rec(f, botsink, g); break;   /* f̄ ∧ g = (f? 0: g) */
    case 6: r = xor_rec(f, g); break;   /* f ⊕ g */
    case 7: r = or_rec(f, g); break;   /* f ∨ g */
    case 8: r = constrain_rec(f, g); break;   /* f ↓ g */
    case 9: r = all_rec(f, g); break;   /* f A g */
    case 10: case 12: r = yes_no_rec(curop, f, g); break;   /* f N g or f Y g */
    case 14: r = diff_rec(f, g); break;   /* f D g */
    case 15: r = exist_rec(f, g); break;
    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)\n",
        id(r), id(f), binopname[curop], id(g), mems − oldmems, rmems − oldrmems, zmems − oldzmems);
  return r;
}
```

107. ⟨ Templates for subroutines 25 ⟩ +≡

```
void attempt_repairs(void);   /* collect garbage or something if there's hope */
```

108. Have you any wool? Yes sir, yes sir.

⟨ Subroutines 7 ⟩ +≡

```
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 = and_exist_rec(f, g, h); break;   /* (f ∧ g) E 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)\n",
        id(r), id(f), ternopname1[curop − 16], id(g), ternopname2[curop − 16], id(h), mems − oldmems,
        rmems − oldrmems, zmems − oldzmems);
  return r;
}
```

109. ⟨ Subroutines 7 ⟩ +≡

```
node *compose_top(node *f)
{
  node *r;
  unsigned long long oldmems = mems, oldrmems = rmems, oldzmems = zmems;
  if (f ≤ topsink) return f;
  if (verbose & 2) printf("beginning␣to␣compute␣%x[%u]:\n", id(f), thevar(f)→timestamp);
  cacheinserts = 0;
  timestamp_changed = 0;
  while (1) {
    r = compose_rec(f);
    if (r) break;
    attempt_repairs();   /* try to carry on */
  }
  if (verbose & (1 + 2)) printf("␣%x=%x[%u]␣(%llu␣mems,␣%llu␣rmems,␣%llu␣zmems)\n",
        id(r), id(f), thevar(f)→timestamp, mems − oldmems, rmems − oldrmems, zmems − oldzmems);
  return r;
}
```

110. 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 110⟩ ≡
  {
    ⟨Make sure the coast is clear 111⟩;
    ⟨Fill buf with the next command, or goto alldone 113⟩;
    ⟨Parse the command and execute it 114⟩;
  }
```

This code is used in section 2.

111. Before we do any commands, it's helpful to ensure that no embarrassing anomalies will arise. For example, a command like 'f0=x1&x2&x3' might necessitate making space for up to three new variables; it would be a nuisance if those attempts failed. (See the *projection* routine.) We also want to check that *timestamp* hasn't reached its maximum possible value.

```
#define debugging 1

⟨Make sure the coast is clear 111⟩ ≡
#if debugging & includesanity
  if (verbose & 8192) sanity_check();
#endif
  if (totalnodes ≥ toobig) ⟨Invoke autosifting 153⟩;
  if (verbose & 1024) show_stats();
  while (1) {
    if (timestamp ≠ −1) {
      ⟨If there are at least three free pages and at least three free nodes, break 17⟩;
    }
    attempt_repairs();
  }
```

This code is used in section 110.

112.

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

⟨Global variables 4⟩ +≡
  char buf[bufsize];   /* our master's voice */
```

113.

```
⟨Fill buf with the next command, or goto alldone 113⟩ ≡
  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 110.

114. 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 114 ⟩ ≡

```
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 146 ⟩; continue;
  case 'C': print_cache(); continue;
  case 'f': ⟨ Parse and execute an assignment to f_k 120 ⟩; continue;
  case 'i': ⟨ Get ready to read a new input file 116 ⟩; continue;
  case 'l': getk; leasesonlife = k; continue;
  case 'm': ⟨ Print a Mathematica program for a generating function 160 ⟩; continue;
  case 'o': ⟨ Output a function 118 ⟩; continue;
  case 'O': ⟨ Print the current variable ordering 119 ⟩; continue;
  case 'p': ⟨ Print a function or its profile 117 ⟩; continue;
  case 'P': print_base(0, 0); continue;   /* P means "print all" */
  case 'q': goto alldone;   /* this will exit the program */
  case 'r': ⟨ Reset the reorder trigger 152 ⟩; continue;
  case 's': ⟨ Swap variable x_k with its predecessor 130 ⟩; continue;
  case 'S':
    if (isdigit(*c)) ⟨ Sift on variable x_k 147 ⟩
    else siftall(); continue;
  case 't': ⟨ Reset tvar 129 ⟩; continue;
  case 'v': getk; verbose = k; continue;
  case 'V': verbose = -1; continue;
  case 'y': ⟨ Parse and execute an assignment to y_k 128 ⟩; continue;
  case '$': show_stats(); continue;
  default: reporterror;
  }
nextcommand: continue;
```

This code is used in section 110.

115. ⟨ 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 */
```

116. 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 116⟩ ≡

```
  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 114.

117. The command '`p3`' prints out the BDD 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 ≥ varsize) { printf("x%d␣is␣out␣of␣range.\n", k); continue; }
```

⟨Print a function or its profile 117⟩ ≡

```
  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], 0);
  }
```

This code is used in section 114.

118. The command '`o3`' outputs the BDD for f_3; the command '`ou3`' outputs it with variable names "unmapped", so that all branches go to variables with higher numbers (in spite of any reordering that has been done). Unmapped BDD output is important for programs such as BDDREAD-COUNT, which count the number of solutions, because those programs need to know how many levels are being crossed at every downward branch.

⟨Output a function 118⟩ ≡

```
  {
    int unmapped = 0;
    if (*c ≡ 'u') c++, unmapped = 1;   /* ou means "output unmapped" */
    getkf;
    sprintf(buf, "/tmp/f%d.bdd", k);
    freopen(buf, "w", stdout);   /* redirect stdout to a file */
    print_function(f[k], unmapped);
    freopen("/dev/tty", "w", stdout);   /* restore normal stdout */
  }
```

This code is used in section 114.

119. ⟨Print the current variable ordering 119⟩ ≡

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

This code is used in section 114.

120. My little finite-state automaton.

⟨ Parse and execute an assignment to f_k 120 ⟩ ≡
```
  getkf; lhs = k;
  passblanks;
  if (*c++ ≠ '=') reporterror;
  ⟨ Get the first operand, p 121 ⟩;
  ⟨ Get the operator, curop 122 ⟩;
second: ⟨ Get the second operand, q 123 ⟩;
third: ⟨ If the operator is ternary, get the third operand, r 124 ⟩;
fourth: ⟨ Evaluate the right-hand side and put the answer in r 125 ⟩;
  ⟨ Assign r to f_k, where k = lhs 126 ⟩;
```
This code is used in section 114.

121. **#define** *checknull*(*p*)
```
    if (¬p) { printf("f%d␣is␣null!\n", k); continue; }
```

⟨ Get the first operand, p 121 ⟩ ≡
```
  passblanks;
  switch (*c++) {
  case 'x': getkv; p = projection(varmap[k]); break;
  case 'f': getkf; p = f[k]; checknull(p); break;
  case 'c': k = *c++ − '0'; if ((k & −2) ≡ 0) p = botsink + k; else reporterror; break;
  case '~': p = topsink; curop = 6; goto second;      /* reduce ¬f to 1 ⊕ f */
  case '.': ⟨ Dereference the left-hand side 127 ⟩; continue;
  default: reporterror;
  }
```
This code is used in section 120.

122. Quantification uses the following conventions:

- A command like '`f1=f2 A x2`' sets $f_1 \leftarrow \forall x_2\, f_2$.
- A command like '`f1=f2 E f3`' where $f_3 = x_4 \wedge x_5$ sets $f_1 \leftarrow \exists x_4\, \exists x_5\, f_2$.
- A command like '`f1=f2&f4 E f3`' with that f_3 sets $f_1 \leftarrow \exists x_4\, \exists x_5\, (f_2 \wedge f_4)$.

⟨ Get the operator, *curop* 122 ⟩ ≡

```
passblanks;
switch (*c++) {
case '&': curop = 1; break;   /* and */
case '>': curop = 2; break;   /* butnot */
case '<': curop = 4; break;   /* notbut */
case '^': curop = 6; break;   /* xor */
case '|': curop = 7; break;   /* or */
case '_': curop = 8; break;   /* constrain */
case 'A': curop = 9; break;   /* forall */
case 'N': curop = 10; break;   /* no */
case 'Y': curop = 12; break;   /* yes */
case 'D': curop = 14; break;   /* diff */
case 'E': curop = 15; break;   /* exists */
case '?': curop = 16; break;   /* if-then-else */
case '.': curop = 17; break;   /* median */
case '[': curop = 0;   /* functional composition */
  if (*c++ ≠ 'y') reporterror;
  if (*c++ ≠ ']') reporterror;
  goto fourth;
case '\n': curop = 7, q = p, c--; goto fourth;   /* change unary p to p ∨ p */
default: reporterror;
}
```

This code is used in section 120.

123. ⟨ Get the second operand, q 123 ⟩ ≡

```
passblanks;
switch (*c++) {
case 'x': getkv; q = projection(varmap[k]); break;
case 'f': getkf; q = f[k]; checknull(q); break;
case 'c': k = *c++ − '0'; if ((k & −2) ≡ 0) q = botsink + k; else reporterror; break;
default: reporterror;
}
```

This code is used in sections 120 and 128.

124. ⟨ If the operator is ternary, get the third operand, r 124 ⟩ ≡

```
passblanks;
if (curop ≡ 1 ∧ *c ≡ '&') curop = 18;      /* and-and */
if (curop ≡ 1 ∧ *c ≡ 'E') curop = 19;      /* and-exists */
if (curop ≤ maxbinop) r = Λ;
else {
  if (*c++ ≠ ternopname2[curop − 16][0]) reporterror;
  passblanks;
  switch (*c++) {
  case 'x': getkv; r = projection(varmap[k]); break;
  case 'f': getkf; r = f[k]; checknull(r); break;
  case 'c': k = *c++ − '0'; if ((k & −2) ≡ 0) r = botsink + k; else reporterror; break;
  default: reporterror;
  }
}
```

This code is used in section 120.

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

⟨ Evaluate the right-hand side and put the answer in r 125 ⟩ ≡

```
passblanks;
if (*c ≠ '\n' ∧ *c ≠ '#') {      /* comments may follow '#' */
reportjunk: c++;
  reporterror;
}
if (curop ≡ 0) r = compose_top(p);
else if (curop ≤ maxbinop) r = binary_top(curop, p, q);
else r = ternary_top(curop, p, q, r);
```

This code is used in section 120.

126. 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).

⟨ Assign r to f_k, where $k = lhs$ 126 ⟩ ≡

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

This code is used in section 120.

127. ⟨ Dereference the left-hand side 127 ⟩ ≡

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

This code is used in section 121.

128. ⟨ Parse and execute an assignment to y_k 128 ⟩ ≡

```
getkv; v = &varhead[varmap[k]];
if (o, ¬v→proj) projection(k);   /* ensure that x_k exists */
passblanks;
if (*c++ ≠ '=') reporterror;
passblanks;
if (*c ≡ '.') c++, q = Λ;
else {
  ⟨ Get the second operand, q 123 ⟩;
  if (o, q ≡ node_(v→proj)) q = Λ;
}
passblanks;
if (*c ≠ '\n' ∧ *c ≠ '#') goto reportjunk;
if (o, v→repl ≠ addr_(q)) {
  p = node_(v→repl);
  if (p) deref(p);
  ⟨ Assign q as the new value of v→repl 102 ⟩;
}
```

This code is used in section 114.

129. 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 BDDs. (Note that *varmap* is not relevant here.)

⟨ Reset *tvar* 129 ⟩ ≡

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

This code is used in section 114.

130. Reordering. Now comes the new stuff, where BDD14 enters the territory into which BDD11 was afraid to tread. Everything is 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 130 ⟩ ≡

```
getkv; v = &varhead[varmap[k]];
if (o, ¬v→proj) projection(k);     /* ensure that x_k exists */
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 114.

131. 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.

⟨ Subroutines 7 ⟩ +≡

```
void reorder_init(void)
{
  var *v, *vup;
  collect_garbage(1);
  totalvars = 0;
  for (v = varhead, vup = Λ; v < topofvars; v++)
    if (v→proj) {
      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;
}
```

132. ⟨ Global variables 4 ⟩ +≡

```
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 */
```

133. 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.

Notice also that the projection function x_u is always solitary; the projection function x_v is always remote. But 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 133⟩ ≡
  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 134⟩;
    ⟨Create a new unique table for x_u and move the remote nodes to it 135⟩;
    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 139⟩;
    ⟨Transmogrify the tangled nodes and insert them in their new guise 140⟩;
    ⟨Delete the lists of solitary, tangled, and hidden nodes 143⟩;
    if (verbose & 2048) printf("␣newbies␣%d,␣change␣%d,␣mems␣(%llu,0,%llu)\n",
          totalnodes − icount + hcount, totalnodes − icount, mems − omems, zmems − ozmems);
    ⟨Swap names, projections, and replacement functions 144⟩;
  }
```

This code is used in section 145.

134. 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 134 ⟩ ≡

```
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 ((pl > topsink ∧ (o, thevar(pl) ≡ v)) ∨ (ph > topsink ∧ (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 ((ql > topsink ∧ (o, thevar(ql) ≡ v)) ∨ (qh > topsink ∧ (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 133.

135. 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 135 ⟩ ≡

```
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 133.

136. ⟨ Global variables 4 ⟩ +≡

```
addr savebase[maxhashpages];   /* pages to be discarded after swapping */
```

137. 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 7 ⟩ +≡

```
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);
}
```

138. 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 7⟩ +≡
```
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 32⟩;
  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);
}
```

139. ⟨Create a new unique table for x_v and move the solitary nodes to it 139⟩ ≡
```
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 133.

140. 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 $g\to lo = h\to lo$ or $g\to hi = h\to hi$.) Similar formulas apply when either g or h lies below level v.

⟨ Transmogrify the tangled nodes and insert them in their new guise 140 ⟩ ≡

```
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);
  if (g ≤ topsink) vg = topofvars; else o, vg = thevar(g);
  if (h ≤ topsink) vh = topofvars; else o, 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 ? g : node_(g→hi), vh > v ? h : 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 133.

141. 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 7 ⟩ +≡

```
node *swap_find(var *v, node *l, node *h)
{
  register int j, k, mask, free;
  register addr *hash;
  register node *p, *r;
  if (l ≡ h) {    /* 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 142 ⟩;
}
```

142. ⟨ Create a newbie and return it 142 ⟩ ≡

```
if (−−free ≤ mask ≫ 4) ⟨ Double the table size and goto restart 32 ⟩;
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 141.

143. ⟨ Delete the lists of solitary, tangled, and hidden nodes 143 ⟩ ≡

```
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 133.

144. Several of the following operations are unnecessary overkill. For example, instead of interchanging *u→proj* with *v→proj*, and using *projection*(*varmap*[*k*]) to access the projection function for x_k, I could leave *u→proj* and *v→proj* unchanged and just say '*projection*(*k*)'. However, the interaction with replacement functions and composition gets tricky, so I've decided to play it safe (for a change): All *repl* and *proj* 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.

⟨ Swap names, projections, and replacement functions 144 ⟩ ≡

```
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→repl;
oo, u→proj = v→proj, u→repl = v→repl;
o, v→proj = j, v→repl = k;
if (v→repl ≠ 0 ∧ (o, v→timestamp ≡ 0)) {
   for (j = v − varhead; j > u − varhead; j−−) o, varhead[j].timestamp = 1;
} else if (v→repl ≡ 0 ∧ (o, v→timestamp ≠ 0) ∧ (v + 1 ≡ topofvars ∨ (o, (v + 1)→timestamp ≡ 0))) {
   for (j = v − varhead; j > u − varhead; j−−) o, varhead[j].timestamp = 0;
}
```

This code is used in section 133.

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

⟨ Subroutines 7 ⟩ +≡
   ⟨ Declare the *swap* subroutine 133 ⟩

146. ⟨Bubble sort to reestablish the natural variable order 146⟩ ≡

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

This code is used in section 114.

147. 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 147⟩ ≡

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

This code is used in section 114.

148. 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 7⟩ +≡

```
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 149⟩;
siftdown: ⟨Explore in the downward direction 150⟩;
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 */
}
```

149. 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 149⟩ ≡

```
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 */
  o, swaps++, swap(u, u→down);
  u = u→down;
}
goto wrapup;
```

This code is used in section 148.

150. ⟨Explore in the downward direction 150⟩ ≡

```
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 148.

151. 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 7 ⟩ +≡

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

152. 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 152 ⟩ ≡

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

This code is used in section 114.

153. ⟨ Invoke autosifting 153 ⟩ ≡

```
{
  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 111.

154. ⟨ Global variables 4 ⟩ +≡

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

155. I should mention a surprising feature of BDD14 that is not a bug: Sometimes a sifting operation can actually *increase* the size of a function, even when only one f_k is defined!

For example, consider $(x_1?\ x_3 \wedge x_4\colon x_2?\ x_3\colon x_4) \wedge x_5$. The profile of this function is $(1,1,2,1,1,2)$; and after swapping $x_4 \leftrightarrow x_5$ it is $(1,1,2,2,1,2)$. But BDD14 does *not* consider this to be a change in the total number of nodes, because all of the projection functions are also implicitly present. When we consider the projection functions x_1, x_2, x_3, x_4, x_5 in addition to the stated function, the profile is $(2,2,3,2,1,2)$ both before and after swapping.

156. 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 BDD 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 7⟩ +≡
  void collect_garbage(int level)
  {
    register int k;
    var *v;
    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;
      ⟨Clear out the time stamps 103⟩;
    }
    if (verbose & 512) printf("collecting␣garbage␣(%d/%d)\n", deadnodes, totalnodes);
    for (v = varhead; v < topofvars; v++) table_purge(v);
  }
```

157. 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.

```
158. ⟨Global variables 4⟩ +≡
  int last_ditch;     /* are we backed up against the wall? */
```

```
159. ⟨Subroutines 7⟩ +≡
  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 6⟩;
      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 */
  }
```

160. Mathematica output. An afterthought: It's easy to output a (possibly huge) file from which Mathematica will compute the generating function.

⟨ Print a Mathematica program for a generating function 160 ⟩ ≡

```
getkf;
math_print(f[k]);
fprintf(stderr, "(generating␣function␣for␣f%d␣written␣to␣%s)\n", k, buf);
```

This code is used in section 114.

161. ⟨ Global variables 4 ⟩ +≡

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

162. ⟨ Subroutines 7 ⟩ +≡

```
void math_print(node *p)
{
  var *v;
  int k, s, ss, t;
  node *q, *r;
  if (¬p) return;
  outcount++;
  sprintf(buf, "/tmp/bdd14-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−−)
      if (v→proj) ⟨ Generate Mathematica outputs for variable v 163 ⟩;
    unmark(p);
  }
  fprintf(outfile, "g%x\n", id(p));
  fclose(outfile);
}
```

163. ⟨ Generate Mathematica outputs for variable v 163 ⟩ ≡

```
{
  t = 0;
  for (k = 0; k < v→mask; k += sizeof(addr)) {
    q = fetchnode(v, k);
    if (q ∧ (q→xref + 1) < 0) {
      t = 1;
      ⟨ Generate a Mathematica line for node q 164 ⟩;
    }
  }
  if (t) v→aux = ++s;    /* this many levels exist below v */
}
```

This code is used in section 162.

164. ⟨ Generate a Mathematica line for node q 164 ⟩ ≡

```
fprintf(outfile, "g%x=Expand[", id(q));
r = node_(q→lo);
ss = (r ≤ topsink ? 0 : thevar(r)→aux);
if (ss < s) {
  if (s ≡ ss + 1) fprintf(outfile, "(1+z)*");
  else fprintf(outfile, "(1+z)^%d*", s − ss);
}
fprintf(outfile, "g%x+z*", id(r));
r = node_(q→hi);
ss = (r ≤ topsink ? 0 : thevar(r)→aux);
if (ss < s) {
  if (s ≡ ss + 1) fprintf(outfile, "(1+z)*");
  else fprintf(outfile, "(1+z)^%d*", s − ss);
}
fprintf(outfile, "g%x]\n", id(r));
```

This code is used in section 163.

165. Index.

⟨ Assign q as the new value of $v\to repl$ 102 ⟩ Used in section 128.
⟨ Assign r to f_k, where $k = lhs$ 126 ⟩ Used in section 120.
⟨ Bubble sort to reestablish the natural variable order 146 ⟩ Used in section 114.
⟨ Build the shadow memory 60 ⟩ Used in section 58.
⟨ Check that p is findable in the unique table 63 ⟩ Used in section 60.
⟨ Check the cache 69 ⟩ Used in section 58.
⟨ Check the command line 3 ⟩ Used in section 2.
⟨ Check the list of free nodes 64 ⟩ Used in section 60.
⟨ Check the list of free pages 70 ⟩ Used in section 58.
⟨ Check the reference counts 66 ⟩ Used in section 58.
⟨ Check the unique tables 68 ⟩ Used in section 58.
⟨ Clear out the time stamps 103 ⟩ Used in section 156.
⟨ Compute the ghost index fields 65 ⟩ Used in section 60.
⟨ Create a new node and return it 31 ⟩ Used in section 26.
⟨ Create a new unique table for x_u and move the remote nodes to it 135 ⟩ Used in section 133.
⟨ Create a new unique table for x_v and move the solitary nodes to it 139 ⟩ Used in section 133.
⟨ Create a newbie and return it 142 ⟩ Used in section 141.
⟨ Create a unique table for variable hv with size 2 28 ⟩ Used in section 24.
⟨ Declare the *swap* subroutine 133 ⟩ Used in section 145.
⟨ Delete the lists of solitary, tangled, and hidden nodes 143 ⟩ Used in section 133.
⟨ Dereference the left-hand side 127 ⟩ Used in section 121.
⟨ Double the cache size 47 ⟩ Used in section 46.
⟨ Double the table size and **goto** *restart* 32 ⟩ Used in sections 31, 138, and 142.
⟨ Downsize the cache if it has now become too sparse 50 ⟩ Used in section 49.
⟨ Downsize the table if only a few entries are left 37 ⟩ Used in section 35.
⟨ Evaluate the right-hand side and put the answer in r 125 ⟩ Used in section 120.
⟨ Explore in the downward direction 150 ⟩ Used in section 148.
⟨ Explore in the upward direction 149 ⟩ Used in section 148.
⟨ Fill *buf* with the next command, or **goto** *alldone* 113 ⟩ Used in section 110.
⟨ Find $(f \wedge g)$ E h recursively 98 ⟩ Used in section 97.
⟨ Find $(f?\ g\colon h)$ recursively 92 ⟩ Used in section 91.
⟨ Find $\langle fgh \rangle$ recursively 94 ⟩ Used in section 93.
⟨ Find $f(y_0, \ldots)$ recursively 105 ⟩ Used in section 104.
⟨ Find f A g recursively 85 ⟩ Used in section 84.
⟨ Find f D g recursively 87 ⟩ Used in section 86.
⟨ Find f E g recursively 83 ⟩ Used in section 82.
⟨ Find f Y g or f N g recursively 89 ⟩ Used in section 88.
⟨ Find $f \downarrow g$ recursively 80 ⟩ Used in section 79.
⟨ Find $f \wedge g$ recursively 74 ⟩ Used in section 73.
⟨ Find $f \wedge g \wedge h$ recursively 96 ⟩ Used in section 95.
⟨ Find $f \vee g$ recursively 76 ⟩ Used in section 75.
⟨ Find $f \oplus g$ recursively 78 ⟩ Used in section 77.
⟨ Generate Mathematica outputs for variable v 163 ⟩ Used in section 162.
⟨ Generate a Mathematica line for node q 164 ⟩ Used in section 163.
⟨ Get ready to read a new input file 116 ⟩ Used in section 114.
⟨ Get the first operand, p 121 ⟩ Used in section 120.
⟨ Get the operator, *curop* 122 ⟩ Used in section 120.
⟨ Get the second operand, q 123 ⟩ Used in sections 120 and 128.
⟨ Global variables 4, 8, 13, 22, 30, 40, 42, 51, 59, 101, 112, 132, 136, 154, 158, 161 ⟩ Used in section 2.
⟨ If the operator is ternary, get the third operand, r 124 ⟩ Used in section 120.
⟨ If there are at least three free pages and at least three free nodes, **break** 17 ⟩ Used in section 111.
⟨ Initialize everything 5, 9, 12, 23, 44, 62 ⟩ Used in section 2.

BDD14