fanf Subscribe

February 12 2008, 21:49

Hashlife

INTRODUCTION

Hashlife is a radically powerful way of computing Conway's Game of Life. It was invented by Bill Gosper who originally described it in his 1984 paper "Exploiting Regularities in Large Cellular Spaces", Physica D (Nonlinear Phenomena) volume 10 pp. 75-80.

Three weeks ago I wrote about
Life In A Register which was a tangent from work I was doing on hashlife.
I have not made much progress on it since then, so I thought I would post what I have done so far here, because it is the most interesting part. It's a literate-programming description of the algorithm, optimised for clarity of exposition, not speed.
This post is (mostly) mechanically generated from the

code I have written so far.

Hashlife uses several cunning techniques:

The universe is represented as a quadtree of nested squares that are powers of two cells on a side. There is an intricate and beautiful recursive function for computing life using this data structure.

It avoids constructing multiple copies of equivalent portions of the quadtree that are duplicated at different places or times by looking up previously-constructed squares in the eponymous hash table. (This technique is known as memoizing or hash-consing.) It also remembers the results of Life computations so that it does not need to repeat them.

Because of its aggressive memoization it is extremely effective at computing Life patterns that have some kind of repetition in space or time. Hashlife can take the same length of time to compute from generation N to generation 2N for any large enough N - it has logarithmic complexity. Its disadvantage is that it is not very fast at handling random Life soup.

There are a few hashlife resources on the web. There is David Bell's implementation, and Tomas Rokicki's implementation is part of the "Golly" Game of Life simulator. He also wrote about

it for Dr Dobb's: "An Algorithm for Compressing Space and Time".
For an example of hashlife's capabilities, it can compute 2^130 generations of the Metacatacryst producing a pattern of 2^232 cells.
I first read about hashlife when working on my previous (more conventional) Life implementation.

CORE DATA STRUCTURE AND ALGORITHM

First, a little bit of notation. Because hashlife can handle exceptionally large computations, we represent generation counts, population counts, display co-ordinates, etc. as multiprecision integers. Powers of two are particularly important, so because of the cost of creating mpz_t objects we pre-allocate a few of them. With any luck this upper bound will be overkill.

I'll also use this array for notation in the comments, so pow2[0] == 1, pow2[1] == 2, pow2[8] == 256, etc.

```
#include <stdbool.h>
#include <gmp.h>

typedef unsigned char byte;

#define LEVEL_MAX 1024

static mpz_t pow2[LEVEL_MAX];

static void
```

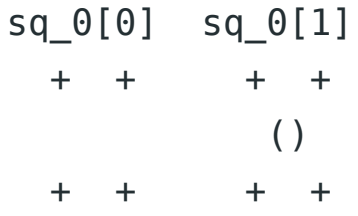
```
        init_pow2(void) {
                mpz_init_set_ui(pow2[0], 1);
                for(unsigned i = 1; i < LEVEL_MAX; i++) {
                        mpz_init(pow2[i]);
                        mpz_mul(pow2[i], pow2[i-1],
pow2[i-1]);
                }
        }
```

Hashlife divides the universe up into a quadtree. A square at level N is pow2[N] cells on a side, and divided into four level N-1 quarters with half as many cells on a side, i.e. pow2[N-1]. We do not need to store the level explicitly since we can keep track of it by counting depth of recursion.

The code is written consistently to handle sub-squares in reading order named by compass points, as shown in this declaration. The quarter pointers are never NULL except in level 0 squares. The calc field can be NULL or a pointer to a level N-1 square; it is filled in lazily. We'll have much more to say about it below.

```
        typedef struct square {
                struct square *nw, *ne,
                              *sw, *se,
                              *calc;
                mpz_t *pop;
        } *square;
```

There are two level 0 squares, each being one cell that is live or dead. We don't represent this state explicitly, but instead just use the identities of the squares, so they don't need initialization. Read "sq_0" as "square level 0".




```
#define NUM_SQ_0 (1 << 1*1)

static struct square sq_0[NUM_SQ_0];
```

This function returns 0 for a dead cell and 1 for a live cell.
The pointer arithmetic is equivalent to (sq == &sq_0[1]).

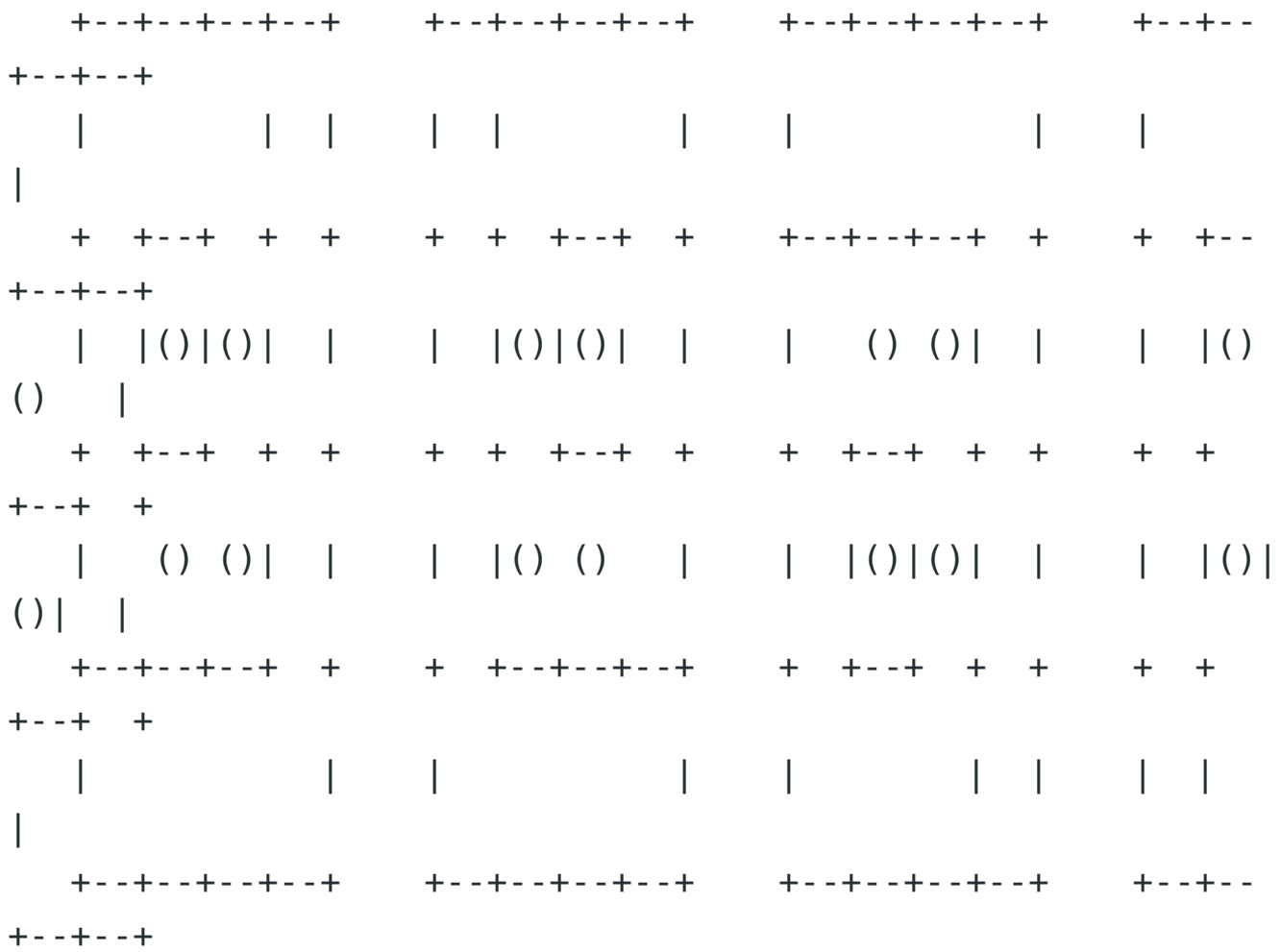
```
static inline bool
alive(square sq) {
        return(sq - sq_0);
}
```

The level 1 squares have a level 0 square (a cell) in each quarter,

so they are 2*2 cells in size and have pow2[2*2] = 16 possible states. The state of each cell is represented by a pointer to sq_0[0] or sq_0[1].

```
#define NUM_SQ_1 (1 << 2*2)

static struct square sq_1[NUM_SQ_1];

static void
init_sq_1(void) {
        for(unsigned i = 0; i < NUM_SQ_1; i++) {
                unsigned nw, ne, sw, se;
                nw = (i & 1) >> 0;
                ne = (i & 2) >> 1;
                sw = (i & 4) >> 2;
                se = (i & 8) >> 3;
                square sq = &sq_1[i];
                sq->nw = &sq_0[nw];
                sq->ne = &sq_0[ne];
                sq->sw = &sq_0[sw];
                sq->se = &sq_0[se];
        }
}
```

A level 2 square similarly comprises four level 1 squares, so it has 16 cells. Level 2 squares are interesting because they are the first squares that are big enough to calculate the next generation. We can calculate the next state of one of the four inner cells based on its eight surrounding cells as shown below:

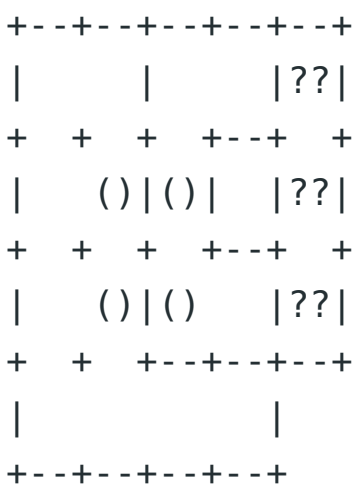


The state of a cell in the next generation is determined by its current state and the number of live neighbours.

```
static bool
life_rule(square nw, square nn, square ne,
          square ww, square cc, square ee,
          square sw, square ss, square se) {
        unsigned count;
        count = alive(nw) + alive(nn) + alive(ne)
              + alive(ww)             + alive(ee)
              + alive(sw) + alive(ss) + alive(se);
        return(count == 2 ? alive(cc) : count == 3);
```

```
        }
```

However we do not have enough information to calculate the next states of the outer cells because our focus is currently too narrow to see how the surrounding pattern might affect them.



All we can say, given a level 2 square, is that we can compute the state after one generation of a level 1 square aligned to the same centre. We will see in a moment how to use this as the basis for calculating the futures of bigger squares.

We pre-compute the result of the calculation on level 2 squares and store it in the square's calc field for re-use later. In larger squares this field is only filled in when necessary, but in the level 2 case it is always filled in so that it acts as a sentinel to save us from explicitly testing for the base case of the recursion.

For reference when reading the double dereferencing,
here's a diagram of how the constituent level 1
sub-squares and level 0 sub-sub-squares
within a level 2 square are laid out.

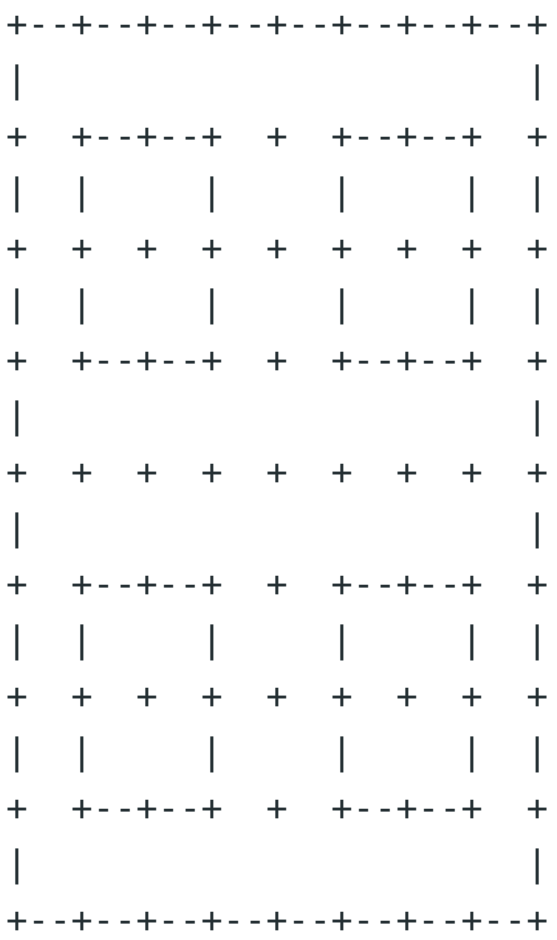
```
   +--+--+--+--+
   |nw ne|nw ne|
nw +  +  +  +  + ne
   |sw se|sw se|
   +--+--+--+--+
   |nw ne|nw ne|
sw +  +  +  +  + se
   |sw se|sw se|
   +--+--+--+--+
```

```
#define NUM_SQ_2 (1 << 4*4)

static struct square sq_2[NUM_SQ_2];

static void
init_sq_2(void) {
        for(unsigned i = 0; i < NUM_SQ_2; i++) {
                unsigned nw, ne, sw, se;
                nw = (i & 0x000F) >> 0;
                ne = (i & 0x00F0) >> 4;
                sw = (i & 0x0F00) >> 8;
                se = (i & 0xF000) >> 12;
                square sq = &sq_2[i];
                sq->nw = &sq_1[nw];
                sq->ne = &sq_1[ne];
                sq->sw = &sq_1[sw];
```

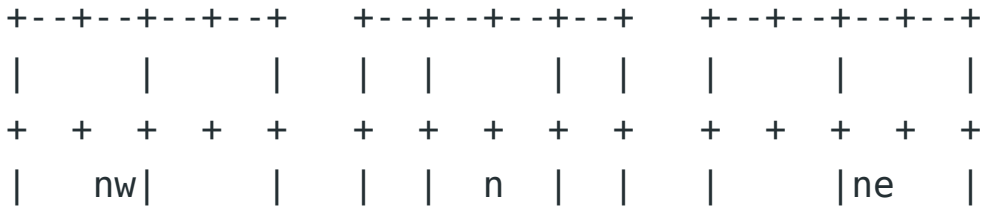
```
                        sq->se = &sq_1[se];
                        // Calculate Life on the four
possible 3x3 groups:
                        unsigned calc = 0;
                        calc |= life_rule(sq->nw->nw, sq->nw-
>ne, sq->ne->nw,
                                          sq->nw->sw, sq->nw-
>se, sq->ne->sw,
                                          sq->sw->nw, sq->sw-
>ne, sq->se->nw) << 0;
                        calc |= life_rule(sq->nw->ne, sq->ne-
>nw, sq->ne->ne,
                                          sq->nw->se, sq->ne-
>sw, sq->ne->se,
                                          sq->sw->ne, sq->se-
>nw, sq->se->ne) << 1;
                        calc |= life_rule(sq->nw->sw, sq->nw-
>se, sq->ne->sw,
                                          sq->sw->nw, sq->sw-
>ne, sq->se->nw,
                                          sq->sw->sw, sq->sw-
>se, sq->se->sw) << 2;
                        calc |= life_rule(sq->nw->se, sq->ne-
>sw, sq->ne->se,
                                          sq->sw->ne, sq->se-
>nw, sq->se->ne,
                                          sq->sw->se, sq->se-
>sw, sq->se->se) << 3;
                        sq->calc = &sq_1[calc];
                }
        }
```

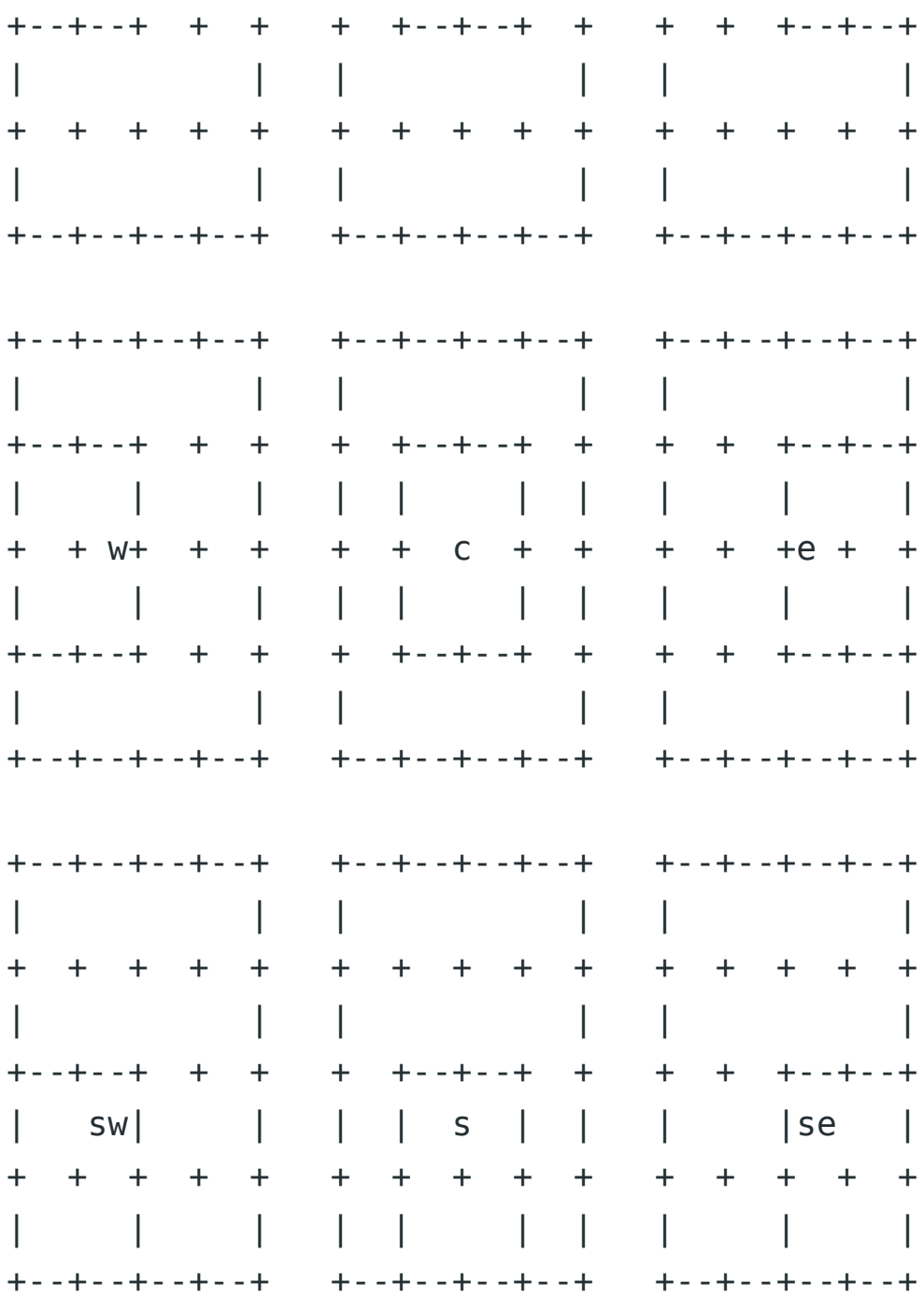
Given the pre-calculated results for level 2 squares, how can use them to calculate the next generation of a level 3 square? We can't

just recurse on its four quarters because that leaves gaps between the results:

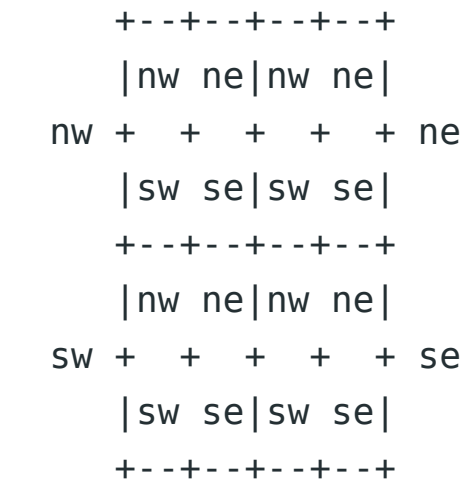


To solve this problem the level 2 sub-squares need to overlap so that their smaller level 1 results abut. We achieve this by re-arranging the level 1 sub-sub-squares to make five extra level 2 sub-squares. We then have a total of nine overlapping level 2 squares. (These diagrams are half-size.)





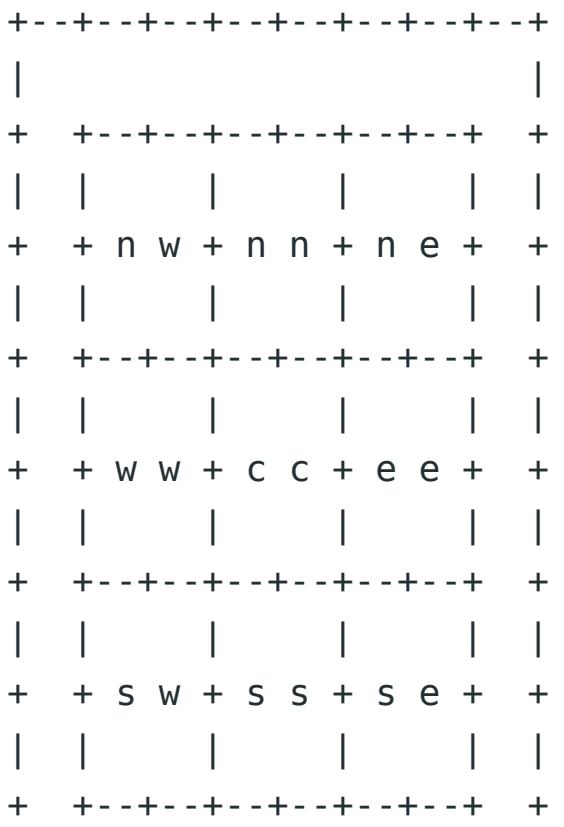
We're going to need these nine sub-squares in a couple of places so let us define helper functions for finding them. The main find() function (which I have not written yet) looks up a square in the hash table based on its four constituent sub-squares, or constructs a new square if it was not found. For reference, here's the sub+subsub diagram again.




```
        static square find(square nw, square ne, square sw,
square se);

        static inline square find_nw(square sq) {
                return(sq->nw);
        }
        static inline square find_nn(square sq) {
                return(find(sq->nw->ne, sq->ne->nw,
                            sq->nw->se, sq->ne->sw));
        }
        static inline square find_ne(square sq) {
                return(sq->ne);
        }
        static inline square find_ww(square sq) {
                return(find(sq->nw->sw, sq->nw->se,
                            sq->sw->nw, sq->sw->ne));
        }
        static inline square find_cc(square sq) {
                return(find(sq->nw->se, sq->ne->sw,
                            sq->sw->ne, sq->se->nw));
        }
        static inline square find_ee(square sq) {
                return(find(sq->ne->sw, sq->ne->se,
```

```
                        sq->se->nw, sq->se->ne));
        }
        static inline square find_sw(square sq) {
                return(sq->sw);
        }
        static inline square find_ss(square sq) {
                return(find(sq->sw->ne, sq->se->nw,
                        sq->sw->se, sq->se->sw));
        }
        static inline square find_se(square sq) {
                return(sq->se);
        }
```

When we calculate the next generation of each of these nine overlapping level 2 sub-squares, we get nine abutting level 1 squares like this:



We can calculate a second generation by combining these into four overlapping level 2 squares and then working out their results. This gives us four abutting level 1 squares as follows. I have labelled each of these new level 1 squares with the four previous level 1 squares that were used to make their parent level 2 squares. The repeated labels indicate where the overlap occurs.

Finally we combine these level 1 squares to make another level 2 square. We could use this to calculate a third generation, but we

do not because after two generations the work we have done starting from a level 3 square is twice as big in every dimension - both space and time - as the level 2 calculation. This gives us a perfect recursive structure, which is the key to the algorithm.

	level 2	level 3	level N
input size	4 * 4	8 * 8	pow2[N] * pow2[N]
output size	2 * 2	4 * 4	pow2[N-1]*pow2[N-1]
generations	1	2	pow2[N-2]

In Life, the "speed of light", the maximum speed that a signal can propagate, is one cell per generation. This is the rate at which the behaviour of surrounding squares can affect more and more of the cells around the edge of our current square as time passes. Eventually after pow2[N-2] generations our focus is too narrow to say anything about a border pow2[N-2] cells wide, and we only know about the middle square pow2[N-1] cells across. (This is why a square's calc field points to a level N-1 square, just like its quarter fields.) Hence, the ziggurat (truncated pyramid) shape formed by hashlife's recursion is the "past light cone" of the result square, that is, it includes all the cells in past generations that can possibly affect the state of the cells in the result.

The calculation function itself is memoized, i.e. it avoids repeating work by storing its result in the square. This makes the overlapping recursive structure feasible, because without the memoization it would lead to enormous amounts of recomputation. Another way of looking at this is lazy evaluation of the calc field, i.e. call-by-need, where we explicitly force evaluation of the calc field by calling calc2(). The

combination of maximal sharing of equivalent squares (enabled by find()'s hash table), memoization of results, and divide-and-conquer recursion is what gives hashlife its amazingly fast logarithmic complexity for patterns that have repeating behaviour.

The memoization is where the pre-computed level 2 calc results act as sentinels, since the test for a memoized result always succeeds at level 2 and terminates the recursion.

```
        static square
        calc2(square sq) {
                // Check for a memo or a sentinel.
                if(sq->calc)
                        return(sq->calc);
                // Find level N-1 sub-squares and calculate
level N-2 results.
                square
                    nw = find_nw(sq), nn = find_nn(sq), ne =
find_ne(sq),
                    ww = find_ww(sq), cc = find_cc(sq), ee =
find_ee(sq),
                    sw = find_sw(sq), ss = find_ss(sq), se =
find_se(sq);
                nw = calc2(nw); nn = calc2(nn); ne =
calc2(ne);
                ww = calc2(ww); cc = calc2(cc); ee =
calc2(ee);
                sw = calc2(sw); ss = calc2(ss); se =
calc2(se);
                // Construct intermediate overlapping level
N-1 squares
                // and calculate their level N-2 results.
```

```
            nw = calc2(find(nw, nn, ww, cc));
            ne = calc2(find(nn, ne, cc, ee));
            sw = calc2(find(ww, cc, sw, ss));
            se = calc2(find(cc, ee, ss, se));
            // Memoize the result.
            sq->calc = find(nw, ne, sw, se);
            return(sq->calc);
        }
```

What if we want to calculate a number of generations that isn't a power of two? We keep exactly the same spatial geometry (relative sizes and positions of input and output squares) as for the full calculation, so that the recursion also has the same shape. The simplest case of calculating fewer generations is zero, in which case we just construct a sub-square centred on the original square. In fact we already have code to do this in find_cc().

In the more general case the desired generation count is between 0 as worked out by find_cc(), and pow2[N-2] as worked out by calc2(). There are three kinds of recursion depending on whether the target falls before, on, or after the midway generation pow2[N-3]. This point in time corresponds to the age of the nine intermediate results in calc2(), which is also a dividing point in its code and in calc1() below.

```
                    recursion in first part or second part
          0 < gen < pow2[N-3]      calc1        find_cc
              gen = pow2[N-3]      calc2        find_cc
    pow2[N-3] < gen < pow2[N-2]      calc2         calc1
```

In calc1() we test the type of recursion for each part separately, and the three cases arise from the way the two tests overlap when cmp == 0. In the second part we have to adjust the generation count so that it is relative to the midway point, and we must restore its value before returning so that it appears unchanged to the caller. (We manipulate its value in-place because that is more efficient than allocating and freeing a local mpz_t with the required value.)

The following code assumes that 0 < gen < pow2[N-2]. Again the base case of the recursion is not explicit. The level argument must not be less than 3, in which case the assumption implies that gen must be 1 which is equal to the halfway point. Therefore no more recursion via calc1() occurs, and the calls to calc2() return immediately because of the level 2 sentinels.

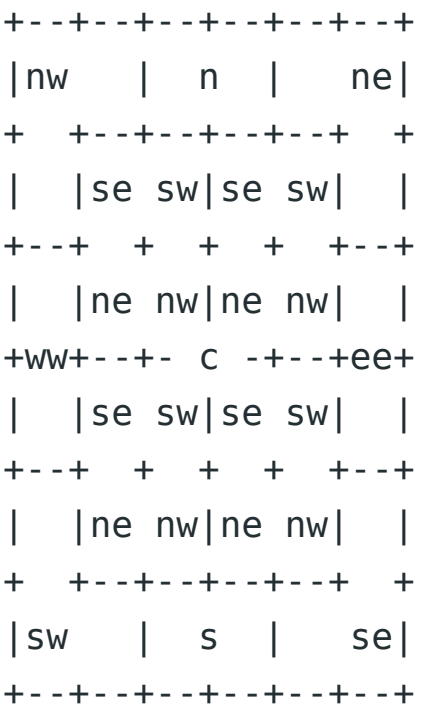
This function is not memoized, but instead relies on calc2() for efficiency.

```
        static square
        calc1(square sq, unsigned level, mpz_t gen) {
                // Number of generations to the midway point.
                mpz_t *half = &pow2[level - 3];
                int cmp = mpz_cmp(gen, *half);
                level -= 1;
                square
                    nw = find_nw(sq), nn = find_nn(sq), ne =
find_ne(sq),
                    ww = find_ww(sq), cc = find_cc(sq), ee =
```

```
find_ee(sq),
                    sw = find_sw(sq), ss = find_ss(sq), se =
find_se(sq);
                if(cmp < 0) {
                        nw = calc1(nw, level, gen);
                        nn = calc1(nn, level, gen);
                        ne = calc1(ne, level, gen);
                        ww = calc1(ww, level, gen);
                        cc = calc1(cc, level, gen);
                        ee = calc1(ee, level, gen);
                        sw = calc1(sw, level, gen);
                        ss = calc1(ss, level, gen);
                        se = calc1(se, level, gen);
                } else {
                        nw = calc2(nw); nn = calc2(nn); ne =
calc2(ne);
                        ww = calc2(ww); cc = calc2(cc); ee =
calc2(ee);
                        sw = calc2(sw); ss = calc2(ss); se =
calc2(se);
                }
                if(cmp > 0) {
                        mpz_sub(gen, gen, *half);
                        nw = calc1(find(nw, nn, ww, cc),
level, gen);
                        ne = calc1(find(nn, ne, cc, ee),
level, gen);
                        sw = calc1(find(ww, cc, sw, ss),
level, gen);
                        se = calc1(find(cc, ee, ss, se),
level, gen);
                        mpz_add(gen, gen, *half);
                } else {
```

In the following code, if we followed the pattern and

called find_cc() where the code above calls calc1(), it would require up to eight allocations. We can reduce this maximum to just four by avoiding the creation of the intermediate results, and instead directly creating the results that find_cc() would return according to the following diagram.




```
                        nw = find(nw->se, nn->sw, ww->ne, cc-
>nw);
                        ne = find(nn->se, ne->sw, cc->ne, ee-
>nw);
                        sw = find(ww->se, cc->sw, sw->ne, ss-
>nw);
                        se = find(cc->se, ee->sw, ss->ne, se-
>nw);
                }
                return(find(nw, ne, sw, se));
        }
```

UNWRITTEN SUPPORT CODE

The code above is the essence of hashlife.
The most important thing it omits is the definition of the find() function,
which serves as the square constructor. It has to implement hash-consing,
i.e. it only ever constructs one copy of a square with a given set of arguments,
to ensure maximal sharing. For serious use it must also implement
garbage collection. I don't know if I can leave that job to the
Boehm-Demers-Weiser conservative collector, but if I do then I'll have to
do something cunning with weak pointers in find()'s hash table.

Maybe I'll write that eventually...

COMPANY

About

News

Help

PRODUCTS

"Share" button

COMMUNITY

Frank

CHOOSE LANGUAGE

ENGLISH

Privacy Policy

User Agreement

Recommendation technologies

Help

v.796