# On Solving Hexadoku and Debugging Recursive Programs with Message Digests of the Data Stack

François Laagel*
Institute of Electrical and Electronics Engineers

## Abstract

Debugging Forth code is difficult, especially when dealing with recursive code. One has to maintain invariants and ensure that the data stack is not unduly modified. This document presents a new technique called *stack digesting* which helps the programmer quickly converge on coding errors. It is based on a cryptographic message digest algorithm that is completely specified in [1]. Although the NIST deprecated this message digest generation mechanism in 2011, it serves our debugging purposes well enough. The idea is based on the insertion of strategically placed probing points in the code being debugged so as to make sure that invariants are actually preserved.

## 1 Problem Statement

Recursion often is the most natural way to express an algorithm when dealing with intrinsically combinatorial problems (see [2]). Elektor Magazine is a publication that caters to electronics enthusiasts of all kinds. Every two months they publish an issue which includes a puzzle called *Hexadoku*. The rules are similar to the traditional Sudoku puzzle. However Hexadoku extends the regular 3 by 3 nature of Sudoku to a 4 by 4 use case. As a result we end up with a 16 by 16 grid instead of the traditional 9 by 9. Each spot in the grid can be any digit expressed in hexadecimal. The usual Sudoku integrity constraints apply; they are simply extended:

- digit uniqueness in a 4 by 4 sub-quadrant.
- digit uniqueness in an horizontal row.
- digit uniqueness in a vertical column.

This kind of puzzle begs for an automated problem solver and one way to approach it is to think of the problem in a three dimensional manner. The grid itself is implemented as a two dimension array of cells. Each cell of the array is interpreted either as a known value (a power of two corresponding to a resolved value) or a bitmask corresponding to the sum of viable alternatives for that spot. This is similar to the notion of "sum over all possibilities", a central tenet of the path integral formulation of quantum theory (see [3]).

The automated solver I implemented works not by looking for a solution but by converging on one by systematic elimination of unviable alternatives. It does so by alternating between phases of inference and speculation until eventually all spots are resolved (have a cell value that is a power of two).

- `infer` goes systematically over the whole grid and updates bitmasks based on resolved spots values, taking into account the constraints defined by the rules of the puzzle. In effect, this greatly reduces the size of the search space.
- `speculate` selects a currently unresolved spot location and recursively explores the space of open possibilities for that spot. It uses feedback supplied by `infer` to detect constraints violations and backtrack when appropriate. An application specific *transaction stack* is used to record all inferred changes to the grid and to undo them when constraints are detected as being violated. A transaction boundary occurs (and is flagged as such) whenever a speculative decision is made. Basically a transaction includes all inferred changes to the grid since the latest speculation.

In essence it works as a greedy minimalization algorithm aiming at reducing the number of unresolved spots values to zero. Its kernel is as outlined in figure 1. Some implementation details need to be elaborated on at this point.

- `rl+`/`rl-` increment and decrement a global variable that holds the current recursion level. The maximum recursion level also is maintained.
- `get-unresolved` selects the first unresolved grid cell for which the number of set bits is minimal but stricly greater than one. This is where the greedy aspect of the algorithm originates from since we always attack the problem from an angle where the number of options is the smallest at any given point in time.

*`f.laagel@ieee.org`

```
: speculate ( -- success-flag )
  rl+                           \ Increment recursion level

  get-unresolved                \ Look for an unresolved spot
  DUP 0= IF INVERT EXIT THEN    \ Problem solved

  DUP @                         \ S: saddr\sval
  \ The list of set bits in TOS indicate the possibilities
  \ for the selected spot. Explore these alternatives.
  16 0 DO
    DUP I 2^n AND IF
      OVER TRUE SWAP tstk-push \ Insert transaction boundary

      OVER I 2^n SWAP
        +ul |visual -ul !       \ Un-logged update-spot

      infer IF                  \ No inconsistencies detected
        RECURSE IF              \ Stop on the 1st solution found
          2DROP UNLOOP TRUE EXIT
        THEN
      THEN

      \ Backtrack up to the last transaction boundary.
      BEGIN tstk-pop UNTIL
      nbt 1+!               \ Increment the number of backtracks

    THEN
  LOOP

  2DROP FALSE                   \ Dead end reached
  rl- ;                         \ Decrement recursion level
```

Figure 1: The Core Speculation Engine

- `tstk-push`/`tstk-pop` take care of handling the transaction stack. `speculate` goes over the alternatives for one selected spot only. Which explains the `16 0 DO ... LOOP` construct. Whenever a speculative decision is made, that change is logged to the the transaction stack under a *boundary* marker. Changes later inferred will also be stacked up so that that they can be undone, should the current speculative decision prove not to be conducive to a workable solution.

- `|visual` updates the on-screen representation of the current solution state. It is stack neutral but uses the *next of stack* as the new grid cell value for the spot pointed to by the *top of stack*.

- `+ul`/`-ul` select and deselect the underline font style in order to display the speculation spots in a way that stands out.

- `infer` will deduce all the consequences of a speculative decision until either a constraint violation is detected or the number of unresolved spots can no longer be decreased (a viable situation).

During the execution of the loop in `speculate`, it is essential that the top two values of the data stack be preserved. They are the cell address of the unresolved spot we are working on and that spot's original superposition of possible states.

I quickly devised a working implementation that ultimately converged on a solution matching the constraints defined by the rules of the game. Yet, it was not entirely satisfying since zeroes–a zero grid cell value indicates an impossible condition for the

```
>speculate at rl 15
              [ 4 , 1 ] <- 5           Cell owned at rl 15
>speculate at rl 16
                [ 4 , 12 ] <- 7
>speculate at rl 17
                 [ 4 , 4 ] <- C
                 [ 4 , 7 ] <- E
                 [ 5 , 7 ] <- 4
                 [ 10 , 7 ] <- 4        Vertical constraint violation
                 [ 10 , 5 ] <- 8
                 >backtrack
                 [ 10 , 5 ] <-
                 [ 10 , 7 ] <-
                 [ 5 , 7 ] <-
                 [ 4 , 7 ] <-
                 [ 4 , 4 ] <-
                 <backtrack
<speculate at rl 17
                >backtrack
                [ 4 , 12 ] <-
                <backtrack
                [ 4 , 1 ] <- E          Cell contents altered at rl 16!
```

Figure 2: Failure at Recursion Level 16

associated spot–started surfacing in the grid and I thought that this situation was not detected early enough, thereby negatively impacting the overall performance of the solver.

So I worked on early detection and avoidance of zero grid cell values. Associated changes to the source code were mostly in the `infer` code. As it turned out, this is where things started going sideways. I had introduced a bug somewhere in the 650 lines or so of code of the solver.

In any sufficiently complex code, these things are bound to happen and I think it is no secret that the classic ways of handling such a conundrum are:

- systematic verification of assumptions. Preconditions can easily be verified by extra tests and references to `ABORT"`. Working code should already have this defense mechanism built-in but if it does not, this is definitely the first step to be taken toward fixing a bug.

- an efficient logging mechanism. This is priceless and also should be an integral part of the original code. Logging should be conditional based on some ad'hoc *debug* vector flag.

- last resort measures such as improved code documentation and/or third party code reviews. These are either time consuming, expensive or both– definitely not practical ways to address the problem at hand.

Post-condition verification is somehow more elusive and the object of this paper is precisely to describe one, with respect to data stack integrity preservation.

After having implemented a reasonable logging mechanism, I realized the code failed to preserve the key invariant `speculate` relies on. It did so at recursion level 16, as illustrated in figure 2. In order to fix this bug rapidly, I introduced the concept of *stack digests*.

```
>speculate at rl 15
              [ 4 , 1 ] <- 5
              >infer 390D7AFB:87768414:F88C2553:C3F8F2A4:4C74CE4F
              <infer 390D7AFB:87768414:F88C2553:C3F8F2A4:4C74CE4F
>speculate at rl 16
               [ 4 , 12 ] <- 7
               >infer 36BF7E72:0BFB296F:E13498C8:D6A8C5DF:C3B6C3D9
               <infer 36BF7E72:0BFB296F:E13498C8:D6A8C5DF:C3B6C3D9
>speculate at rl 17
                [ 4 , 4 ] <- C
                >infer F0812B19:93C42F48:2C610131:FAD0D5D1:24E108A3
                [ 4 , 7 ] <- E
                [ 5 , 7 ] <- 4
                [ 10 , 7 ] <- 4
                [ 10 , 5 ] <- 8
                <infer 36BF7E72:0BFB296F:E13498C8:D6A8C5DF:C3B6C3D9
```

Figure 3: `infer` code is the culprit

## 2 Proposed Solution and Proof of Concept

The original concept was proposed on the *Forth2020* Facebook group and was well received by some of its most respected contributors. [4] proposed a stack checking mechanism that only covers stack depth changes. Later on, [5] developed a complete testing framework for ANS94 compliance. To this day, it remains a reference tool for most testers.

Stack digesting covers both depth and actual stack contents but it does not try to perform stack effect characterization at all. It is a debugging aid only meant to be used as an integrity checking tool.

Basically, a stack digest is just what its name suggests: a cryptographic message digest of the state of the data stack (or a subset of it) as it is when sampled or verified–in this paper support for automated verification is not addressed; visual inspection of the logging output is required for this concept to be of any use. The API is restricted to a single word:

> `SDIGEST` ( i*x u – i*x ) Prints a cryptographic digest of the contents of the data stack, omitting the topmost $u$ cells. The algorithm used for producing this digest is implementation defined.

A quick survey of commonly off the shelf available hashing algorithms revealed those most generally agreed upon were crytographic signatures. Among them, it turned out that the easiest to implement and the best documented one was SHA1. [6] provides a detailed pseudo-code description for SHA1.

Once equipped with this new technology, we are now in a position to instrument the application code and to quickly converge on the problem's root cause (figure 3).

The inference code is about 200 lines long but its primary routine is `reduceall` (see figure 4). Through additional code instrumentation, it was determined that an extraneous `2DROP` reference in `reduce4x4` was responsible for this unwarranted data stack corruption.

```
: reduceall ( -- failure-flag )
  reduce4x4 IF                  \ Constraint violated
    TRUE EXIT
  THEN

  16 0 DO
    I get-horiz-mask IF         \ Constraint violated
      UNLOOP TRUE EXIT
    THEN
    ( S: new-possibly-zero-mask ) I SWAP set-horiz-mask IF
      UNLOOP TRUE EXIT
    THEN

    I get-vert-mask IF          \ Constraint violated
      UNLOOP TRUE EXIT
    THEN
    ( S: new-possibly-zero-mask ) I SWAP set-vert-mask IF
      UNLOOP TRUE EXIT
    THEN

  LOOP
  FALSE ;
```

Figure 4: The `reduceall` word

## 3 Status and Further Work

An actual implementation for `SDIGEST` and the underlying SHA1 message digest generation is available at [7]. It has been validated against well known test vectors on 64 and 32 bit cell targets, regardless of their endianness. This framework has been successfully used to fix a nasty bug in the inference code of my unpublished proprietary *Hexadoku* automated solver.

The concept could be further extended by having a dedicated digest stack and manually coded verification checkpoints. Each entry on the digest stack would be the output of the SHA1 message digest code (5 cells). This would most likely require some extension of the API and is left as an exercise to the interested reader.

## 4 Conclusion

This article was written in the hope that stack digesting could become a useful tool in the Forth programmer's toolbox. It provides a convenient synthetic overview of the state of the data stack, which can be very deep indeed.

Recursion is a powerful technique which allows the developer to formulate solutions to complex problems in very simple terms. However, its use is generally frowned upon in the context of embedded systems software development because stack utilization is basically unpredictable, especially when dealing with heavily data driven algorithms.

## References

[1] D. Eastlake 3rd, P. Jones
*RFC 3174: US Secure Hash Algorithm 1 (SHA1)*
The Internet Society, September 2001.

https://www.rfc-editor.org/rfc/rfc3174

[2] Donald L. Kreher, Douglas R. Stinson
*COMBINATORIAL ALGORITHMS Generation, Enumeration and Search*
*Chapter 4, Backtracking Algorithms*
CRC Press, 2019.

[3] Markus Pössel
*The sum over all possibilities: The path integral formulation of quantum theory*
Einstein Online, 2006.
https://www.einstein-online.info/en/spotlight/path_integrals/

[4] Ulrich Hoffmann
*Stack checking - A debugging aid*
euroFORML Conference Proceedings, 1991.
http://www.euroforth.org/ef91/hoffmann.pdf

[5] John Hayes S1I
*Core ANS94 Test Harness*
Online contents, November 27, 1995.
http://www.forth200x.org/tests/ttester.fs

[6] National Security Agency (original designers)
*SHA-1*
Wikipedia.org, various contributors.
https://en.wikipedia.org/wiki/SHA-1

[7] François Laagel
*SHA-1 sample code for GNU Forth 0.7.3 or SwiftForth 3.7.9*
Online contents, August 15, 2023.
https://github.com/forth2020/frenchie68/blob/main/sdigest-generic.4th