SuDoku Solver Case Study: from specification to RVM-Forth (part I)

Angel Robert Lynas, Bill Stoddart

October 4, 2005

Abstract

A project is underway to formulate a development cycle from B — suitably augmenting its implementation language B0 with reversibility constructs — to a coded implementation in the reversible target language RVM-Forth[3, 4] with translation schemas defined for this final stage. This paper describes the first phase of a case study using the puzzle SuDoku to investigate possible ways of fleshing out such a development cycle. We adopt an experimental approach, using a relatively simple specification as a springboard for what a generated code implementation might look like, and explore correspondences between the specification and implementation.

1 Introduction

1.1 Background and Terminology

The ancient Japanese puzzle SuDoku is of course no such thing; it's American and only a few decades old. Its basic form is a 9×9 grid of squares (which we refer to as a "board", not entirely defensibly) divided into nine 3×3 subsections, hereinafter called "sectors". A puzzle consists of a partly filled-in board which must be filled in such that every row, column and sector contains all the digits 1 to 9.

The structure of the puzzle and its solutions leads naturally to a set-based model, wherein the constraints on a square's value and the properties of a solution can be readily expressed.

The target programming language is based on ANS Forth, and runs in a Reversible Virtual Machine developed by Bill Stoddart [3, 4] with an embedded set implementation built on work by Frank Zeyda [5]. The language will be referred to as RVM-Forth; the defining feature which concerns us is a guard/ choice set of constructs, described in [2], whereby a non-deterministic reversible choice (written as CHOICE) can be made from elements in a set. The guard construct takes a flag from the stack, and if this is false, reverses to the last non-deterministic choice, choosing another forward path. Should there be no choices left, the previous CHOICE is revisited. On running out of options, a `ko` prompt is given to signal this to the user (for instance, if a puzzle turns out to have no solution).

Variables can be declared as reversible, in which case their earlier values are restored on reversal; thus the (important) state obtaining at the time of the CHOICE can be reinstated. The full code for the implementation is provided in appendix C.

1.2 Objectives

The overall context of our research is to investigate the formal development of reversible programs, using a modified version of the B Method[1]. The B language has an exact mathematical description, so that programs written in it can be subject to formal logical analysis (a theorem prover is an important part of any B development environment). B presents a user with (at least) two levels of the B language, which are respectively a specification language (highly expressive but not implementable) and an implementation language. A developer writes both a specification and implementation of a program, and is obliged to show that the implementation satisfies the specification.

Our aim, over several of these reports, is to produce a complete B development cycle of a simple solver, from abstract B specification through to an RB0 implementation (this being our reversible version of the B implementation language).

The RB0 code will compile to RVM-Forth, and we hope to gain some insights into how to optimise that mapping by seeing how various applications can be programmed in RVM-Forth itself. We are particularly interested in set-based representations of data and automatic backtracking, because both of these lend themselves to the logical analysis which is at the heart of the B method. [3]

2 Data Model

The basic requirement is for a mapping from each assigned square of the board to its value, this being the current board; and for the remaining squares, a mapping to a set of their available values. The squares can be indexed sequentially, or by row-column co-ordinates, the latter proving simpler in most areas. We therefore define XY as the set of integers 0..8, and thence a square is a pair (row, column) of type $XY \times XY$, the cartesian product[1] of XY with itself. For convenience, the type of a square is defined as:

$$\mathit{SQUARE} \mathrel{\widehat{=}} (0..8) \times (0..8)$$

The current board is a function from squares to 1..9 — the integer set defined as DIGIT. Initially a partial function, the solution sees it become a total function (and trivially a surjection). So defining a Boolean *solved*, we can specify a variable *board*, beginning thus:

$$\begin{array}{l} \mathit{board} \in \mathit{SQUARE} \nrightarrow \mathit{DIGIT} \wedge \\ \mathit{solved} = \mathit{TRUE} \Rightarrow \mathit{board} \in \mathit{SQUARE} \twoheadrightarrow \mathit{DIGIT} \wedge \ldots \end{array}$$

The specification variable *board* becomes the RVM-Forth variable BOARD. There remains a criterion for validity which each square must fulfil, of course. Complementary to this and used in the RVM-Forth implementation is the function from blank squares to their possible values:

$$\mathit{POSSIBLE} \in \mathit{SQUARE} \nrightarrow \mathbb{P}(\mathit{DIGIT})$$

2.1 Constraint Zone

A further requirement is the notion of a Constraint Zone for a given square, which is the union of its row, column, and sector. The shaded area in fig 1 around square **S** is its constraint zone. The row and column are simply specified; row 3, for instance, is the set

[1]A brief explanation of set operations used in the paper is given in appendix A.

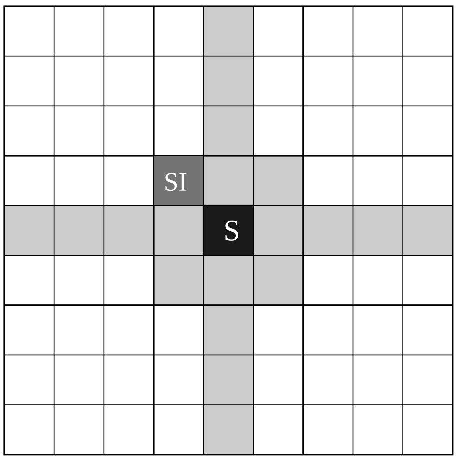

Figure 1: The Constraint Zone for square S

of pairs from (3,0) to (3,8), or the cartesian product of {3} and XY. A function can be defined as a constant *row* to encapsulate this, with properties:

$$\begin{aligned} & row \in XY \rightarrow \mathbb{P}(SQUARE) \wedge \\ & \forall\, rr.(rr \in XY \Rightarrow row(rr) = \{rr\} \times XY) \end{aligned} \tag{1}$$

The sector can be generated in a number of ways; the method used here is to map each square (via a function) to a “sector index” — the square at the top left-hand side of the sector in question (that labelled **SI** in fig 1). The set of sector squares can then be generated orthogonally from these.

Having obtained the constraint zone for a blank square, its relational image with BOARD will yield the subset of *DIGIT* the square *cannot* be assigned. The complement, i.e. *DIGIT* minus these values, will be paired with the square in the initialisation of POSSIBLE. The implementation operation AVAILABLE, which generates these values, is examined below.

2.2 Solution

There remains the rest of the solution specification; while the final board must be a total function, each of its squares must also be valid — that is, the value should not appear elsewhere in its constraint zone. Defining this last as *czone*:

$$czone(r, c) \mathrel{\widehat{=}} row(r) \cup col(c) \cup sector(sectorindex(r, c))$$

A “valid square” function can be defined from a square and a board to a boolean, the salient part of the definition being:

$$\begin{aligned} & is\_valid\_square(rr, cc, bd) = TRUE \Leftrightarrow \\ & bd(rr \mapsto cc) \notin bd[\![\ czone(rr, cc)\ ]\!]) \end{aligned}$$

When *solved* becomes true (having been initialised to false), then for all the squares in *board*, *is_valid_square* is true; it should also be true for every assigned square in partially-filled boards — from which the solution condition would follow. In the code version, this is ensured by the assignment mechanism in any case; illegal assignments would not be possible.

3 Method

An implementation based on the above data model can give us an idea what the final generated product might look like. From the model, an algorithm for a sequential, iterative modus operandi suggests itself; we present the outline followed by some more detailed points. The BOARD and POSSIBLE sets have been initialised at this point: for row r, column c and values v_i, their elements have the forms:

$$\begin{array}{l} \text{BOARD}: \ ((r,c),v) \\ \text{POSSIBLE}: \ ((r,c),\{v_1,v_2,\ldots,v_n\}) \end{array}$$

1. Extract set of most constrained blank squares, along with their values (this is a subset of POSSIBLE).
2. Choose (*non-reversibly*) a square from the domain of this set. Remove this entry from POSSIBLE.
3. Choose a value from those available; this must be a reversible CHOICE. Create a pair from the square and value, and add it to BOARD.
4. Update the blanks in the constraint zone of the square, i.e. remove the value just assigned from their available sets.
5. If any of the resulting sets are empty, the board is now non-viable; in this case, we must backtrack to step 3, restoring state along the way.
6. Otherwise, update POSSIBLE itself with these new values.
7. Repeat until board full.

3.1 Algorithm expansion

1. Concentrating on the most constrained blanks simply seems most logical; should a wrong choice be made, there will be fewer subsequent attempts to work through. More sophisticated techniques would be able to reduce the available values by other comparisons with the current state; our naïve version does not apply all possible constraints, but instead allows the backtracking mechanism to take the strain.
2. The above being the case, it's likely that more than one square will be returned, so one must be chosen to work with. A point which bears stressing is that, if a solution exists at all from this stage, it can be found from *any* of these squares. Faster from some than others, perhaps, though there is no way of knowing which. So:
   (a) The choice may as well be random.
   (b) The choice *must not* be reversible.

   The latter point may well not be obvious at first glance (it can be tempting to assume that any choice should be a CHOICE). However, if none of the values from the chosen square lead to a viable board, reversing will simply cause another square from the set (if any) to be chosen; there's a strong likelihood that this will also fail (needlessly duplicating a fruitless search) *and* will quite possibly close off backward paths to points from which an actual solution could be found.

   In fact the usual upshot (from observation) of trying a reversible square choice is that a handful of squares in mutually exclusive constraint zones end up dealing out the same slightly larger handful of values in an apparently never-ending set of nested loops. This is an example of inappropriate use of CHOICE, since there is no division into “wrong” and “right” squares, and therefore nothing to be gained from trying another one on failure.

3. The choice of value, on the other hand, is clearly critical: probably only one will lead to a solution.; this should be the only reversible CHOICE in the program. If no valid number exists, then a previous assignment must have been wrong.

4. The assigned value is now no longer available to those blanks constrained by the current square, so it must be removed from all of their “possible” assignment sets (a subset of POSSIBLE itself).

5. This should always leave at least one remaining possible value for a square; an empty set here indicates we cannot find a solution given this assignment. This is the test for the reversibility guard, which will provoke backtracking.

6. The action guarded is updating the POSSIBLE set (described in more detail below).

4 Implementation

By virtue of the direct availability of set declarations and operations, many of the data model specifications and algorithm operations translate quite naturally into RVM-Forth, allowing for the postfix conversion. A simple example is the row generator *row()* — recall the function specification in (1) above on page 3, which becomes the following definition:

```
: GENROW ( n -- n.n.*.P )
   INT { , } XY PROD ;
```

The first line is the name and a comment with the stack effect; translating as “integer in, set of integer pairs out”, as the specification says. `INT { }` is one way of creating an integer set, and here the comma between the curly braces allocates space for whatever is on top of the stack, within that set. This gives us `{n}` ; `XY` puts the set of column numbers on the stack while `PROD` generates the cartesian product of these two sets. The issue of garbage collection for such anonymous dynamically-created sets is addressed in appendix B. On its own, GENROW looks like this in action:

```
5 GENROW .SET
{(5,0),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(5,7),(5,8)} ok
```

The union of this with the outputs of GENCOL and GENSECTOR form the set of squares in the constraint zone, returned by GENCZONE. With the square (3,2) on the stack (single line of output split for document):

```
DUP .PAIR CR GENCZONE .SET  (3,2)
{(0,2),(1,2),(2,2),(3,0),(3,1),(3,2),(3,3),
(3,4),(3,5),(3,6),(3,7),(3,8),(4,0),(4,1),
(4,2),(5,0),(5,1),(5,2),(6,2),(7,2),(8,2)}ok
```

Part of the initialisation involves the set POSSIBLE, wherein each blank square is paired with its available values: generation of the latter is performed by AVAILABLE. As described above in section 2.1, what is required is the set of digits in DIGIT which are *not* in the already assigned squares within the constraint zone of the square (r, c) in question; this is specifiable in set notation as:

$$\mathit{DIGIT} - \mathit{board}(\!|\ \mathit{czone}(r, c)\ |\!)$$

Converted for postfix and stack adjustment, it maps exactly to RVM-Forth:

```
: AVAILABLE ( n.n.* -- n.P )
   GENCZONE BOARD SWAP IMAGE DIGIT SWAP \ ;
```

Using the same square as the last example, with a sample puzzle loaded:

```
DUP .PAIR CR AVAILABLE .SET  (3,2)
{1,2,5,8,9}ok
```

4.1 The Update procedure

A more detailed look at this is now presented, it being the section where the reversibility guard comes into play. A boolean variable called VALID acts as a flag for the guard, initialised to *true*. Arguments are the square and the value just assigned to it. Outline:

1. Generate the subset of POSSIBLE which needs updating. This is simply done by finding the constraint zone of the square with GENZONE, and performing a domain restriction (the word `<|` in the definition below: described in appendix A) on POSSIBLE. If the resulting set is not empty, the loop now builds a new set of this type.

2. (Begin loop) For each blank square in this set, remove the assigned value from its set of available values, where present. If the resulting set is empty, set VALID to *false* and exit loop. Otherwise, add the new pair (square, remaining possible values) to the set being built.

3. (End loop) Test VALID: if false, instigate reversal to previous CHOICE (and make another choice). Otherwise, use function override (`<+`) to update POSSIBLE with the new sets of values for the affected squares.

The RVM-Forth code to achieve this is shown below. ASSIGNED is a variable holding the last assigned value, and UNPAIR leaves the first and second elements in that order on the stack.

```
1.  : UPDATE-POSSIBLE ( n.n.*  n -- )
2.     to ASSIGNED GENCZONE POSSIBLE <|
3.     DUP ?{} NOT IF                   ( test for non-empty set )
4.        INT INT PROD INT POW PROD {
5.           DUP CARD 0 DO
6.              DUP I @ELEMENT          ( index through elements )
7.              UNPAIR ASSIGNED SUBTRACT-ELEMENT
8.              DUP ?{} IF FALSE to VALID LEAVE ELSE |->P,S , THEN
9.           LOOP } VALID --> POSSIBLE SWAP <+ to POSSIBLE  THEN DROP ;
```

Line 4 This is the RVM postfix way of specifying a pair comprising a pair of integers and a set of integers (i.e. a square plus set of values)

$$((r, c), \{v_1, v_2, \ldots, v_n\})$$

Line 8 Having subtracted the assigned value, we now check the remaining set with the empty set test. In theory, the guard might go directly after this, but reversing from the middle of a set-building operation is incompatible with the reversibility mechanism; a boolean is used so the guard can be outside.

Line 9 The symbol for the guard is `-->`, here testing VALID. The symbol is cognate with the General Substitution Language's $\Longrightarrow$, normally associated with an IF...THEN construct in a B specification[2]. Here and in the proposed RB0 language, it functions as a "naked" guard, prompting backtracking on failure.

5 Performance

In terms of raw speed, running in a virtual machine atop a subsystem for sets (albeit an efficient one) is unlikely to be optimal. Instead we balance the simplicity of the development using a naïve heuristic against the number of tries the program needs to solve a problem designated "very hard", or "fiendish" (anything less challenging requires little if any backtracking).

While the choice of square could simply take the first one encountered (using ELEMENT), there is a non-backtracking random choice available called PCHOICE, and along with a reversible RANDOM-CHOICE for the value assignment, this gives a variety of possible paths for the program to follow. Sometimes a solution will be found very quickly even for the hardest puzzles in this way. The average range for puzzles encountered so far is 150-500 attempted assignments. The unconstrained search space for assigning about 50 squares from 9 potential values for each one is not considered further here.

6 Further Work

The initial approach here has been to attack both ends of the problem first to gain an idea of what a complete development should encompass. While refinement from the abstract machines will often use the usual techniques, certain differences will be apparent: in the data model, sets need not be refined away as they are directly implementable; also the reversible computations introduce certain differences in refinement methods (and associated proofs), outlined in [6]. Issues thrown up by refinement of this case study will therefore require investigation.

6.1 Code generation: stack vs locals?

The code generation stage is in very early infancy as yet; translation schemas to support this have been begun, but certain questions arise. Forth, RVM or otherwise, is a stack-based language, and much of its operational simplicity derives from not having to declare and handle local variables for basic operations. B, on the other hand, is in the tradition of variable-manipulation languages, and this provides an uncomfortable meeting-point for the two modes.

Provision of an explicit stack at the specification level would cause more problems than it would solve, leaving two alternatives. Ideally, we would aim for a transparent "under the hood" translation system, whereby appropriate use was made of the Forth stack from a standard local-using specification. Initially, however, translation will (relatively) simply

[2]Such programming-style constructs are "syntactic sugar" for the underlying GSL notation.

map B locals to RVM locals, less than optimal though this will prove; meanwhile a reliable way of optimising the translation must be investigated.

The combination of top-down approach from specification and bottom-up approach from code will provide illumination from two aspects for the development of a middle stage, hypothetical as yet, an *implementation-level* specification language closely based on the existing B0.

References

[1] Jean-Raymond Abrial. *The B Book*. Cambridge University Press, 1996.

[2] W J Stoddart. Efficient reversibility with Guards and Choice. In M A Ertl, editor, *18th EuroForth*, 2002. Available from: http://www.complang.tuwien.ac.at/anton/euroforth2002/papers/bill.rev.ps.gz.

[3] W J Stoddart. Using Forth in an Investigation into Reversible Computation. In P Knaggs and M A Ertl, editors, *19th EuroForth*, 2003.

[4] W J Stoddart. RVM-Forth, a Reversible Virtual Machine: User Manual. In M A Ertl, editor, *19th EuroForth*, 2004. Available from http://dec.bournemouth.ac.uk/forth/euro/ef04/stoddart04.pdf or http://www.scm.tees.ac.uk/formalmethods/index.php.

[5] W J Stoddart and F Zeyda. Implementing sets for reversible computation. In A ERTL, editor, *18th Euroforth, Technical University of Vienna*. 2002.

[6] F Zeyda, W J Stoddart, and S E Dunne. The Refinement of Reversible Computations. In T Muntean and K Sere, editors, *2nd International Workshop on Refinement of Critical Systems*, 2003. Available from www.esil.univ-mrs.fr/ spc/rcs03/rcs03.

A Set operations

Many set operations are provided "out of the box" with RVM-Forth, and more advanced ones can be built up with little trouble. For instance, from two sets (of arbitrary types), the set of all possible pairs (cartesian product) can be generated with PROD:

```
SAVOURY SWEET .SET .SET
{chocolate,fruit,honey} {cheese,fish,onion} ok
SWEET SAVOURY PROD .SET
{(chocolate,cheese),(chocolate,fish),(chocolate,onion),
(fruit,cheese),(fruit,fish),(fruit,onion),
(honey,cheese),(honey,fish),(honey,onion)} ok
```

Tasty. As usual, the order of the stack parameters is the same as for the infix operator: $S \times T$ becomes `S T PROD`.

Working with relations and functions is also straightforward. We define a simplistic telephone directory (a relation from strings to integers) to work with, called PHONE, which looks like this:

```
CR PHONE .SET
{(Bill,2673),(Frank,4611),(Keerthi,4611),(Michelle,4611),
(Rob,4611),(Steve,2657)}
```

An operation used in the text is domain restriction (the word `<|`), in which a set of values from the domain is used to extract only those pairs which have a first value in that set — the result being returned as another set. For instance, with a query set dynamically constructed:

```
STRING { " Michelle" , " Keerthi" , } PHONE <|  ok.
.SET {(Keerthi,4611),(Michelle,4611)}ok
```

A relational image is really just the range of this result, though it's normally implemented as a separate operation. The set notation $S(\!|\ U\ |\!)$, where U is a subset of the domain of S, becomes `S U IMAGE` in RVM-Forth:

```
PHONE STRING { " Michelle" , " Keerthi" , } IMAGE ok.
.SET  {4611}ok
```

Which is the range of the previous result. Range restriction is the mirror image of domain restriction (the word `|>`); notice the stack parameters are the other way round to reflect the ordering of the infix operator:

```
PHONE INT { 4611 , } |>  ok.
.SET  {(Frank,4611),(Keerthi,4611),(Michelle,4611),(Rob,4611)}ok
```

A certain overcrowding is becoming apparent. Some rehousing later, the outdated phone numbers can be overwritten with function override. The word `<+` is defined as:

```
: <+ ( s1:x.P s2:x.P -- s3:x.P  where s3 = s1 <+ s2 )
DUP DOM ROT <<| \/ ;
```

Using `<<|`, which is domain subtraction (the complement of domain restriction), this removes the pairs from s1 which are due to be updated, then unions the remainder with s2. To update our phones, we construct a set UPDATES:

```
{(Frank,2680),(Rob,3719)}
```

and invoke function override to effectively overwrite those pairs whose first value matches one of the first values in the update set. Thus:

```
PHONE UPDATES <+ to PHONE ok
CR PHONE .SET
{(Bill,2673),(Frank,2680),(Keerthi,4611),(Michelle,4611),
(Rob,3719),(Steve,2657)}ok
```

This is of course used in the SuDoku program to update the POSSIBLE set with reduced sets of values for the affected squares only.

B Garbage Collection

As might be imagined such wanton creation of arbitrary sets has huge potential for garbage creation; this is automatically collected during reverse execution, but not necessarily otherwise. However, the potential-value capabilities of the RVM allow for the provision of a wrapper which ensures garbage collection for this sort of program. The details are covered in [4], but the results of using these capabilities are shown here with the aid of a diagnostic tool called Heapwatch (© Frank Zeyda). On starting the RVM itself:

```
HW-STATS HeapWatch: Statistical Information:
Current memory in use: 936 bytes (0 KB)
Number of calls to malloc(): 40
Number of calls to calloc(): 0
Number of calls to realloc(): 33 (24 ret. same + 9 diff. address)
Largest memory allocation: 56 bytes in file setkernel.c, line 252.
Average allocation size: 26 bytes
Peak memory utilisation: 936 bytes (0 KB) + 2625 KB for HeapWatch.
```

After a single run of the solver without garbage collection invoked, the situation is as below — followed immediately by another run and the memory report.

```
HW-STATS HeapWatch: Statistical Information:
Current memory in use: 211816 bytes (206 KB)
Number of calls to malloc(): 38819
Number of calls to calloc(): 0
Number of calls to realloc(): 30506 (19868 ret. same + 10638 diff. address)
Largest memory allocation: 344 bytes in file setkernel.c, line 211.
Average allocation size: 44 bytes
Peak memory utilisation: 211816 bytes (206 KB) + 2625 KB for HeapWatch.

    ( ******** another puzzle solved here ********** )

HW-STATS HeapWatch: Statistical Information:
Current memory in use: 427324 bytes (417 KB)
Number of calls to malloc(): 56596
Number of calls to calloc(): 0
Number of calls to realloc(): 43834 (29158 ret. same + 14676 diff. address)
Largest memory allocation: 344 bytes in file setkernel.c, line 211.
Average allocation size: 44 bytes
Peak memory utilisation: 427324 bytes (417 KB) + 2625 KB for HeapWatch.
ok..
```

Clearly some garbage is being left behind, and would continue to build up. Using the `<TRY S CUT>` construct to wrap the program, however, allows it to run, print (or store) its output, and then garbage associated with the run is collected. From a similar cold start to the first quoted above, a run now leaves the system in this situation:

```
HW-STATS HeapWatch: Statistical Information:
Current memory in use: 936 bytes (0 KB)
Number of calls to malloc(): 38819
Number of calls to calloc(): 0
Number of calls to realloc(): 30506 (19866 ret. same + 10640 diff. address)
Largest memory allocation: 344 bytes in file setkernel.c, line 211.
Average allocation size: 44 bytes
Peak memory utilisation: 211816 bytes (206 KB) + 2625 KB for HeapWatch.
```

So the memory allocated by the frequent calls to `malloc()` or `realloc()` has now all been reclaimed.

C RVM-Forth Implementation Code

```
( ========== Declarations and initial Board ==================== )

1 9 .. VALUE DIGIT
0 8 .. VALUE XY
NULL VALUE_ BOARD        ( Reversible variable )
NULL VALUE_ POSSIBLE     ( Reversible variable )
1 VALUE LOOPS 1 VALUE TRIES ( Bookkeeping )
8 VALUE COLI 8 VALUE ROWI 0 VALUE ASSIGNED
TRUE VALUE_ VALID ( Set to false if square left with no domain )

( Generalised board-builder; assumes 81 numbers loaded on stack)
: BUILD-BOARD ( n TIMES 81 --  )
   8 to ROWI
   INT INT PROD INT PROD {
      BEGIN ROWI -1 >
      WHILE
         8 to COLI
         BEGIN COLI -1 >
         WHILE
            DUP 0= NOT
            IF ROWI COLI |->I,I SWAP |->P,I ,
            ELSE DROP
            THEN
            COLI 1- to COLI
         REPEAT
         ROWI 1- to ROWI
      REPEAT } to BOARD ;

( ================ Utilties; zone generation etc ================)

: UNPAIR ( x1.x2.* -- x1 x2 )
   DUP FIRST SWAP SECOND ;

: GENROW ( n -- n.n.*.P )
   INT { , } XY PROD ;

: GENCOL ( n -- n.n.*.P )
   XY SWAP INT { , } PROD ;

( Here, nr is the row number, and nc isn't.
  The output is the row & col of the sector index )
: SECTORINDEX ( nr nc -- n n )
   DUP 3 MOD - SWAP
   DUP 3 MOD - SWAP ;

: GENSECTOR ( nr nc -- n.n.*.P )
   SECTORINDEX
   DUP 2 + .. SWAP
   DUP 2 + .. SWAP PROD ;

( Find the constraint zone for a particular square )
: GENCZONE ( n.n.* -- n.n.*.P )
   UNPAIR DUP
   GENCOL  ROT ROT OVER
   GENROW  ROT ROT
   GENSECTOR  \/ \/ ;

( Now we find the values a blank square can take )
: AVAILABLE ( n.n.* -- n.P )
   GENCZONE BOARD SWAP IMAGE DIGIT SWAP \ ;

( ============================================================ )
( Pretty(ish)-print subsystem. Can safely be ignored )

0 VALUE ELEMINDEX NULL VALUE BOARDSIZE

: VLINE 124 EMIT ;
: LINE VLINE CR ." +---------+---------+---------+" CR  ;

( Convert co-ord pairs to scalar square numbers)
: CONVERTBOARD ( n.n.*.n.P -- )
   INT INT PROD {
      DUP CARD 0 DO
         DUP I @ELEMENT
         UNPAIR SWAP UNPAIR SWAP 9 * +
         SWAP |->I,I ,
      LOOP
   } NIP ;

: .BOARD ( -- )
   CONVERTBOARD
   0 to ELEMINDEX DUP CARD to BOARDSIZE
   81 0 DO ELEMINDEX BOARDSIZE < IF
         DUP ELEMINDEX @ELEMENT
      ELSE DUP 0 @ELEMENT
      THEN
      I 27 MOD 0=
      IF  LINE VLINE ELSE I 9 MOD 0=
         IF  VLINE CR VLINE ELSE I 3 MOD 0=
            IF VLINE
            THEN
         THEN
      THEN
      DUP FIRST I = IF
         SPACE SECOND . ELEMINDEX 1+ to ELEMINDEX
      ELSE 3 SPACES DROP THEN
   LOOP LINE DROP CR ;
( ==================== End of print system ================== )

( Build set of squares and their possible values )
: INIT-POSSIBLE ( -- )
   XY XY PROD BOARD DOM \
   INT INT PROD INT POW PROD {
      DUP CARD 0 DO
         DUP I @ELEMENT
         DUP AVAILABLE |->P,S ,
      LOOP
    } to POSSIBLE DROP ;
```

```
( Get the next squares from the most constrained -- returns a
  subset of POSSIBLE. We cheat a bit by using the set ordering
  to find the lowest card in the range of POSSIBLE )
: GETSQUARES ( n -- n.n.*.n.P.*.P)
   0 (: VALUE N :)
   POSSIBLE DUP RAN ELEMENT CARD to N
   INT INT PROD INT POW PROD {
      DUP CARD 0 DO
         DUP I @ELEMENT
         DUP SECOND CARD N =
         IF , ELSE DROP THEN
      LOOP
   } 1LEAVE ;

( Picks next square and its availables (an element of POSSIBLE)
  to send to assign; subtract it from POSSIBLE )
: NEXTUP ( n.n.*.n.P.*.P -- n.n.*.n.P.* )
   PCHOICE DUP POSSIBLE SWAP
   SUBTRACT-ELEMENT to POSSIBLE ;

( Assigns from square and set of values a single value, adding
  pair to Board; leaves square and value separately )
: ASSIGN ( n.n.*.n.P.* -- n.n.*  n )
   UNPAIR RANDOM-CHOICE TRIES 1+ to TRIES
   2DUP |->P,I BOARD SWAP ADD-ELEMENT to BOARD ;

( Remove assigned from the domain of each square in the constraint
  zone of the last assigned square -- update by func override. Should
  not attempt to update when no blanks need updating )
: UPDATE-POSSIBLE ( n.n.*  n -- )
   to ASSIGNED GENCZONE POSSIBLE <|
   DUP ?{} NOT IF
      INT INT PROD INT POW PROD {
         DUP CARD 0 DO
            DUP I @ELEMENT
            UNPAIR ASSIGNED SUBTRACT-ELEMENT
            DUP ?{} IF FALSE to VALID LEAVE ELSE |->P,S , THEN
         LOOP } VALID --> POSSIBLE SWAP <+ to POSSIBLE  THEN DROP ;

( ============ Run and step-through facilities ================ )
: START  ( -- )
   NULL to BOARD NULL to POSSIBLE
   1 to TRIES 1 to LOOPS
   32 WORD LOAD-FILE BUILD-BOARD INIT-POSSIBLE ;

: STEP  ( -- )
    GETSQUARES NEXTUP
    ASSIGN UPDATE-POSSIBLE LOOPS 1+ to LOOPS ;

: SOLVE ( -- n.n.*.n.*.P )
   BEGIN
      BOARD CARD 81 <
   WHILE
      STEP
   REPEAT  ;
```

```
( Takes filename after word, e.g.: TRY-TO-SOLVE S1 )
: TRY-TO-SOLVE ( -- ) CR
   START SOLVE BOARD .BOARD  TRIES . ." TRIES IN "
   LOOPS . ." LOOPS. " CR ;

( Garbage collecting wrapper for above )
.( TRY <filename> runs the puzzle )
: TRY
   <CHOICE
       <TRY TRY-TO-SOLVE CUT>
     []
       CR
   CHOICE> ;
```