A Clash Course in Solving Sudoku

Gergő Érdi

https://unsafePerform.IO/

Haskell Symposium 2025

16th October 2025.

RetroClash

Orientation

Sudoku: Combinatorial puzzle

Richard Bird 2006.: Derivation of a nice Sudoku solver in Haskell

FPGA: Field-programmable gate array

Clash: Compile Haskell to FPGA configuration

Orientation

Sudoku: Combinatorial puzzle

***Richard Bird 2006.*:** Derivation of a nice Sudoku solver in Haskell

FPGA: Field-programmable gate array

Clash: Compile Haskell to FPGA configuration

This pearl: A Sudoku solver FPGA described in Clash, based on Bird's implementation

Sudoku

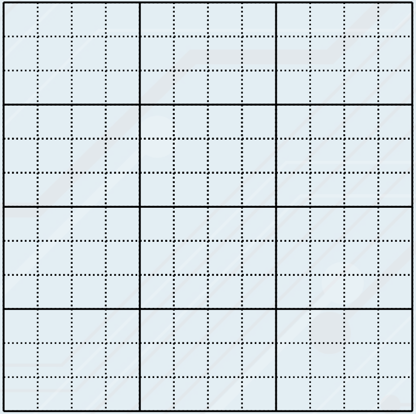
	2		9		8			
8	7				1		5	4
5		6	4				1	
		2					9	5
9	4					8		
	8				4	5		3
1	3		2				8	6
			3		7		2	

(a) If there was a problem ...

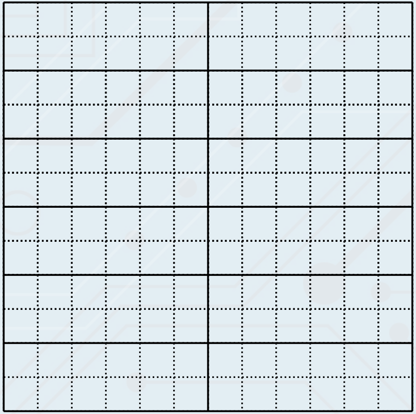
4	2	1	9	5	8	6	3	7
8	7	3	6	2	1	9	5	4
5	9	6	4	7	3	2	1	8
3	1	2	8	4	6	7	9	5
7	6	8	5	1	9	3	4	2
9	4	5	7	3	2	8	6	1
2	8	9	1	6	4	5	7	3
1	3	7	2	9	5	4	8	6
6	5	4	3	8	7	1	2	9

(b) ...yo I'd solve it

(n, m)-Sudoku



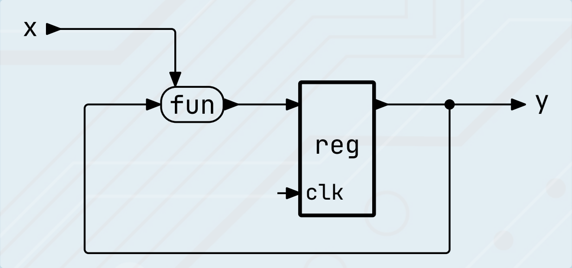
(a) (3,4)-Sudoku



(b) (2,6)-Sudoku

Clash in one slide

Register transfer-level description of synchronous digital circuits:




```
type Signal :: Domain → Type → Type
register :: a → Signal dom a → Signal dom a
instance Applicative (Signal dom)

y :: Signal dom (Unsigned 5)
y = register 19 (fun <$> x <*> y)   -- Note the recursion guarded by register
```

A program to solve Sudoku (Richard Bird 2006.)

Classic backtracking algorithm:

- For each cell, keep track of candidate values.
- Prune canditate values by propagating known values.
- If can't prune more, try guesses.
- If some cell has no more possible values, we're blocked; backtrack.

A program to solve Sudoku (Bird 2006.) rewritten

```
type Sudoku :: Nat → Nat → Type

sudoku :: (Alternative f) ⇒ Sudoku n m → f (Sudoku n m)
sudoku grid
    | blocked   = empty
    | complete  = pure grid
    | changed   = sudoku pruned
    | otherwise = asum [sudoku grid' | grid' ← expand grid]
  where
    -- blocked, complete, changed, pruned ...

expand :: Sudoku n m → [Sudoku n m]
```

A program to solve Sudoku (Bird 2006.) rewritten

```
type Sudoku :: Nat → Nat → Type

sudoku :: Sudoku n m → Maybe (Sudoku n m)
sudoku grid
    | blocked   = empty
    | complete  = pure grid
    | changed   = sudoku pruned
    | otherwise = asum [sudoku grid' | grid' ← expand grid]
  where
    -- blocked, complete, changed, pruned ...

expand :: Sudoku n m → [Sudoku n m]
```

Towards hardware: fixed-size representations

```
type Vec :: Nat → Type → Type   -- From Clash standard library
newtype Matrix n m a = FromRows (Vec n        (Vec m        a))
newtype Grid   n m a = Grid     (Matrix n m (Matrix m n a))

type BitVector :: Nat → Type   -- From Clash standard library
newtype Cell   n m = Cell {cellBits :: BitVector (n * m)}
type    Sudoku n m = Grid n m (Cell n m)
```

Single-valued `Cells` for pruning

```
newtype Mask n m = Mask {maskBits :: BitVector (n * m)}
```

```
cellMask :: Bool → Cell n m → Mask n m
cellMask isSingle (Cell c) =
  if isSingle then Mask c else mempty
```

We pass `isSingle` as a parameter because we want to avoid repeatedly recomputing it for the same `Cell`

Propagation: pruning is a monoidal action

```haskell
instance Semigroup (BitMask n m) where
  Mask m1 ◇ Mask m2 = Mask (m1 .|. m2)
instance Monoid (BitMask n m) where
  mempty = Mask zeroBits
```

```haskell
act :: Mask n m → Bool → Cell n m → Cell n m
act (Mask m) isSingle c =
  if isSingle then c else Cell (cellBits c .&. complement m)
```

Groups: the constraint structure of Sudoku

$$\mathbf{type}\ \mathtt{Group}\ \mathtt{n}\ \mathtt{m}\ \mathtt{a} = \mathtt{Vec}\ (\mathtt{n} * \mathtt{m})\ \mathtt{a}$$

A `Grouping` witnesses the isomorphism between a `Grid` and a collection of nm groups:

$$\mathbf{type}\ \mathtt{Groups}\quad \mathtt{n}\ \mathtt{m}\ \mathtt{a} = \mathtt{Vec}\ (\mathtt{n} * \mathtt{m})\ (\mathtt{Group}\ \mathtt{n}\ \mathtt{m}\ \mathtt{a})$$

$$\mathbf{type}\ \mathtt{Grouping}\ \mathtt{n}\ \mathtt{m}\quad = \forall\ \mathtt{a}.\ \mathtt{Grid}\ \mathtt{n}\ \mathtt{m}\ \mathtt{a} \leftrightarrow \mathtt{Groups}\ \mathtt{n}\ \mathtt{m}\ \mathtt{a}$$

$$\mathbf{data}\ \mathtt{a} \leftrightarrow \mathtt{b} = \mathtt{Iso}\ \{\mathtt{embed} :: \mathtt{a} \to \mathtt{b}, \mathtt{project} :: \mathtt{b} \to \mathtt{a}\}$$

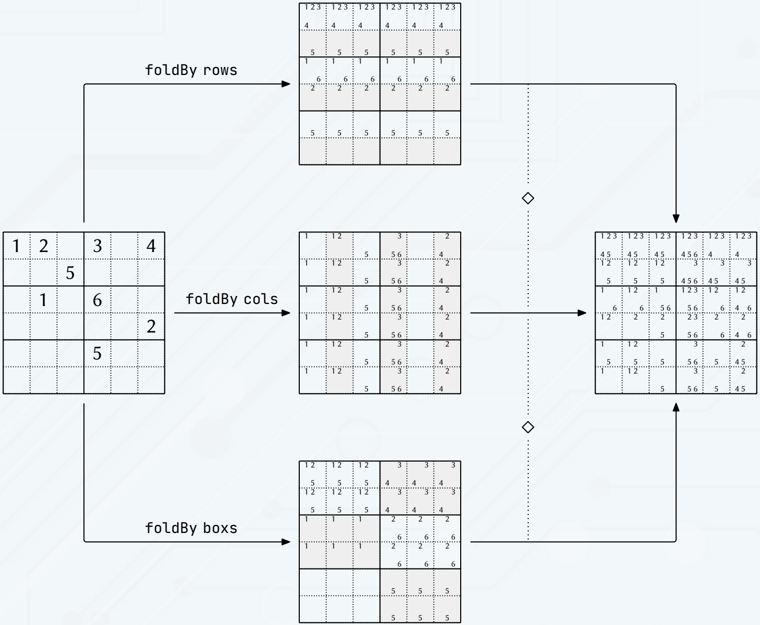
$$\mathtt{rows}, \mathtt{cols}, \mathtt{boxs} :: \mathtt{Grouping}\ \mathtt{n}\ \mathtt{m}$$

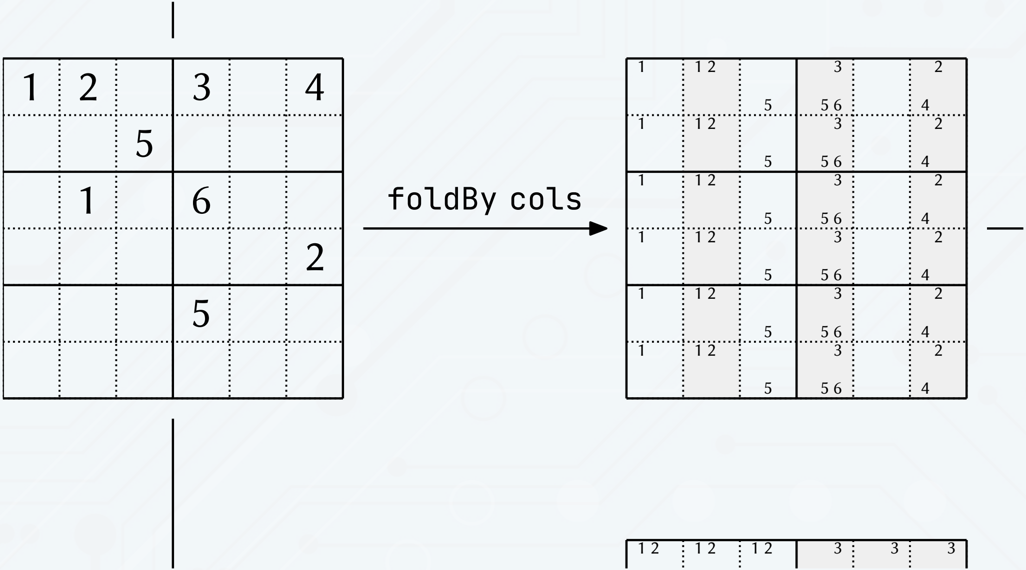
Folding over Groups

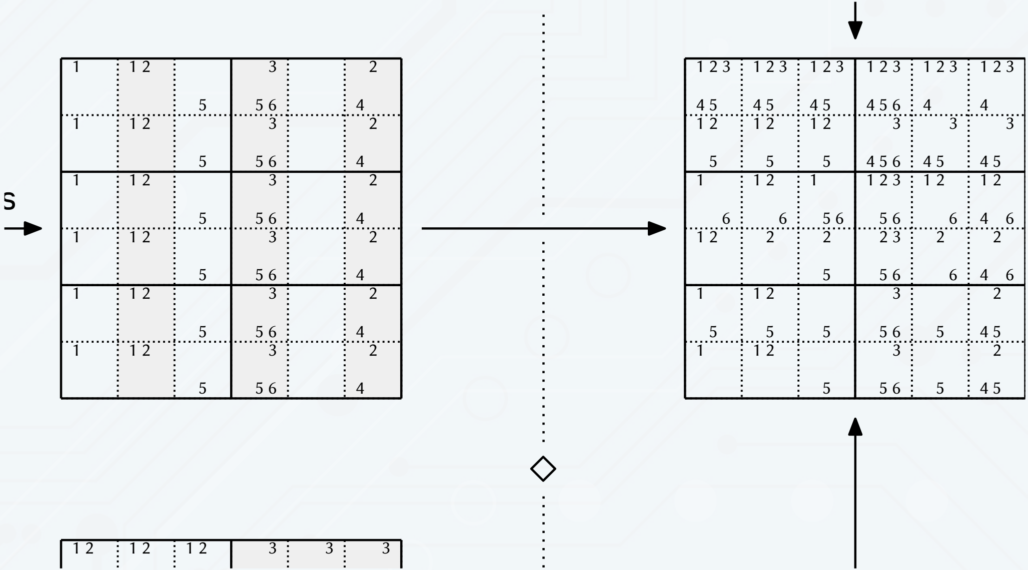
Each cell belongs to three groups: a row, a column, and a box. To fold groupwise, we compute the combination of all values in each group. Then for each position, we combine the three combined results.

$$\mathtt{foldGroups} :: \forall\ \mathtt{a\ n\ m}.\ (\mathtt{Monoid\ a}) \Rightarrow \mathtt{Grid\ n\ m\ a} \rightarrow \mathtt{Grid\ n\ m\ a}$$
$$\mathtt{foldGroups} = \mathtt{foldBy\ rows} \diamond \mathtt{foldBy\ cols} \diamond \mathtt{foldBy\ boxs}$$
$$\quad \mathbf{where}$$
$$\qquad \mathtt{foldBy\ grp} = \mathtt{project\ grp} \circ \mathtt{fmap}\ (\mathtt{repeat} \circ \mathtt{fold}) \circ \mathtt{embed\ grp}$$

For the case of $\mathtt{foldGroups}\ @(\mathtt{Mask\ n\ m})$, this is constraint propagation!


foldBy rows
foldBy cols
foldBy boxs
◇
◇


foldBy cols


S
1 | 1 2 | 5 | 3 5 6 | | 2 4
1 | 1 2 | 5 | 3 5 6 | | 2 4
1 | 1 2 | 5 | 3 5 6 | | 2 4
1 | 1 2 | 5 | 3 5 6 | | 2 4
1 | 1 2 | 5 | 3 5 6 | | 2 4
1 | 1 2 | 5 | 3 5 6 | | 2 4
1 2 | 1 2 | 1 2 | 3 | 3 | 3
1 2 3 4 5 | 1 2 3 4 5 | 1 2 3 4 5 | 1 2 3 4 5 6 | 1 2 3 4 | 1 2 3 4
1 2 5 | 1 2 5 | 1 2 5 | 3 4 5 6 | 3 4 5 | 3 4 5
1 6 | 1 2 6 | 1 5 6 | 1 2 3 5 6 | 1 2 6 | 1 2 4 6
1 2 | 2 | 2 5 | 2 3 5 6 | 2 6 | 2 4 6
1 5 | 1 2 5 | 5 | 3 5 6 | 5 | 2 4 5
1 | 1 2 | 5 | 3 5 6 | 5 | 2 4 5

Two-way branching

$$\texttt{expand} :: \texttt{Sudoku n m} \to [\texttt{Sudoku n m}]$$

When we need to guess, we can `expand` the current grid into just two:

$$\texttt{expand} :: \texttt{Sudoku n m} \to (\texttt{Sudoku n m}, \texttt{Sudoku n m})$$

Select a currently ambiguous `Cell`:

$$\texttt{splitCell} :: \texttt{Cell n m} \to (\texttt{Cell n m}, \texttt{Cell n m})$$

- The *first guess* clears all but one bits of the chosen `Cell`
- The *continuation* is the grid without that one bit

Hardware-friendly `splitCell`

$$\texttt{splitCell} :: \texttt{Cell n m} \rightarrow (\texttt{Cell n m}, \texttt{Cell n m})$$
$$\texttt{splitCell (Cell c)} = (\texttt{Cell least}, \texttt{Cell rest})$$
$$\quad \textbf{where}$$
$$\qquad \texttt{least} = \texttt{leastSetBit c}$$
$$\qquad \texttt{rest} \;\; = \texttt{c .\&. complement least}$$
$$\texttt{leastSetBit} :: \texttt{BitVector n} \rightarrow \texttt{BitVector n}$$
$$\texttt{leastSetBit x} = \texttt{x .\&. negate x}$$

(See Brent Yorgey's Fenwick tree pearl in JFP)

Hardware-friendly `splitCell`

```
splitCell :: Cell n m → (Cell n m, Cell n m)
splitCell (Cell c) = (Cell least, Cell rest)
  where
    least = leastSetBit c
    rest  = c .&. complement least
leastSetBit :: BitVector n → BitVector n
leastSetBit x = x .&. negate x
```

(See Brent Yorgey's Fenwick tree pearl in JFP)

N.b. `rest == 0` exactly means `isSingle c`!!

Recursion

Our definition of sudoku now looks like this:

```
sudoku :: Sudoku n m → Maybe (Sudoku n m)
sudoku grid
    | blocked   = empty
    | complete  = pure grid
    | changed   = sudoku pruned                      -- Tail-recursive
    | otherwise = sudoku guess <|> sudoku cont       -- Non-tail-recursive
  where
    ...
```

This is not a valid Clash function, since it describes a circuit of unbounded size: all recursive occurrences of sudoku contain a copy of the circuit.

Opening up the recursion

```
data Step n m = Blocked
              | Complete
              | Progress (Sudoku n m)
              | Stuck    (Sudoku n m) (Sudoku n m)

solve :: Sudoku n m → Step n m
```

We will do one `Step` (i.e. full propagation of all immediate constraints) per clock cycle.

Explicit stack (list model)

```
sudoku grid = go grid []
  where
    go :: Sudoku n m → [Sudoku n m] → Maybe (Sudoku n m)
    go grid st = case solve grid of
      Blocked
        | (grid' : st') ← st → go grid' st'          -- Pop
        | otherwise          → empty
      Complete               → pure grid
      Progress pruned        → go pruned st
      Stuck guess cont       → go guess (cont : st)  -- Push
```

Stack implemented with RAM

On real hardware, we use synchronous block RAM to implement the stack:

$$\textbf{data}\ \mathtt{MemCmd\ n\ d} = \mathtt{Write}\ (\mathtt{Index\ n})\ \mathtt{d} \mid \mathtt{Read}\ (\mathtt{Index\ n})$$
$$\mathtt{ram} :: \mathtt{Signal\ dom}\ (\mathtt{Maybe}\ (\mathtt{MemCmd\ n\ d})) \to \mathtt{Signal\ dom}\ (\mathtt{Maybe\ d})$$

Each ”stack frame” is just a `Sudoku n m` board.

Guessing always creates one more unambiguous `Cell`, and we never add more candidates to `Cells`, only ever remove them: max stack depth is fixed at n^2m^2, the size of the board.

$$\textbf{type}\ \mathtt{StackPtr} = \mathtt{Index}\ (\mathtt{n} * \mathtt{m} * \mathtt{m} * \mathtt{n})$$

The solver keeps a `StackPtr` state, updated when `Blocked` (to pop) or `Stuck` (to push).

I/O

Our Sudoku solver is a standalone device usable over a serial link.

Problem descriptions are read from serial input cell by cell; solution is sent over serial with whitespace formatting:

```
. 2 . | 9 . 8 | . . .
8 7 . | . . 1 | . 5 4
5 . 6 | 4 . . | . 1 .
------+-------+------
. . 2 | . . . | . 9 5
. . . | . . . | . . .
9 4 . | . . . | 8 . .
------+-------+------
. 8 . | . . 4 | 5 . 3
1 3 . | 2 . . | . 8 6
. . . | 3 . 7 | . 2 .
```

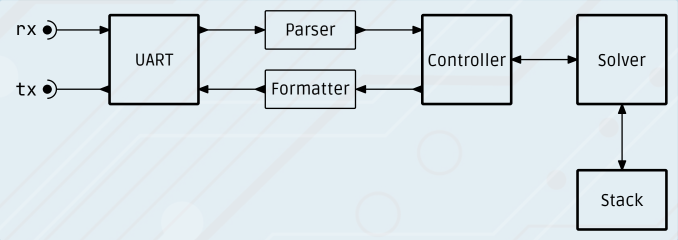
```
4 2 1  9 5 8  6 3 7
8 7 3  6 2 1  9 5 4
5 9 6  4 7 3  2 1 8

3 1 2  8 4 6  7 9 5
7 6 8  5 1 9  3 4 2
9 4 5  7 3 2  8 6 1

2 8 9  1 6 4  5 7 3
1 3 7  2 9 5  4 8 6
6 5 4  3 8 7  1 2 9
```

Serial interface

Using the `clash-protocols` library, we can compose the parser and formatter with the Sudoku solver's controller.



`serialize` is the UART equivalent of software Haskell's `interact` function:

$$\texttt{serialize @9600} \left(\texttt{parser} \rhd \texttt{controller @3 @3} \rhd \texttt{formatter}\right)$$
$$:: \texttt{Signal dom Bit} \rightarrow \texttt{Signal dom Bit}$$

Results

- **Software testing**: Haskell for pure functions (for unit testing) and Clash's simulator (for integration testing)
- Most (3,3)-Sudoku boards solved in **less than 100 cycles**, some "hard" puzzles take thousands of cycles
- **Real hardware target**: Xilinx XC7A50T at 100 MHz (tens of μs solving time for hard puzzles)
- Vivado synthesizes it in about 10 minutes, uses 9,298 / 32,600 logic cells for (3,3) solver, 25,978 cells for (3,4).

https://unsafePerform.IO/clash-sudoku