Mini Sudoku Constraints

Constraint satisfaction via Kan extensions

Simon Frost

Table of contents

Introduction

This vignette demonstrates how FunctorFlow's left Kan extensions (Σ) can express constraint satisfaction problems. We model a 4×4 mini-Sudoku as a categorical diagram where:

- Objects represent cell digits, unit relations (rows, columns, blocks), and violation counts
- Σ (left Kan extensions) aggregate cell digits into each unit, producing digit histograms
- Morphisms count constraint violations (duplicate digits per unit)

This is a direct Julia port of the Python FunctorFlow `sudoku_demo.py`, showing how the same categorical patterns work in both languages.

Setup

```
using FunctorFlow
using Random
```

The 4×4 Sudoku Structure

A 4×4 Sudoku has 16 cells arranged in a 4×4 grid with four 2×2 blocks. Each row, column, and block must contain exactly the digits 0–3 with no repeats.

```
const BOARD_SIZE = 4
const BLOCK_SIZE = 2
const NUM_DIGITS = 4
const NUM_CELLS = BOARD_SIZE * BOARD_SIZE

# Cell indexing: row-major, 0-based to match Python
cell_id(row, col) = row * BOARD_SIZE + col
```

```
cell_id (generic function with 1 method)
```

Building Unit Relations

The three constraint units (rows, columns, blocks) define incidence relations — each unit maps to the set of cell indices it contains. These are the "along" functors for our Kan extensions.

```
function build_unit_relations()
    rows = Dict(
        Symbol("row_$r") => [cell_id(r, c) for c in 0:BOARD_SIZE-1]
        for r in 0:BOARD_SIZE-1
    )
    cols = Dict(
        Symbol("col_$c") => [cell_id(r, c) for r in 0:BOARD_SIZE-1]
        for c in 0:BOARD_SIZE-1
    )
    blocks = Dict{Symbol, Vector{Int}}()
    idx = 0
    for r in 0:BLOCK_SIZE:BOARD_SIZE-1
        for c in 0:BLOCK_SIZE:BOARD_SIZE-1
            blocks[Symbol("block_$idx")] = [
```

```
                cell_id(r + dr, c + dc)
                for dr in 0:BLOCK_SIZE-1
                for dc in 0:BLOCK_SIZE-1
            ]
            idx += 1
        end
    end
    (rows=rows, cols=cols, blocks=blocks)
end

relations = build_unit_relations()
println("Row 0 cells:   ", relations.rows[:row_0])
println("Col 2 cells:   ", relations.cols[:col_2])
println("Block 3 cells: ", relations.blocks[:block_3])
```

```
Row 0 cells:   [0, 1, 2, 3]
Col 2 cells:   [2, 6, 10, 14]
Block 3 cells: [10, 11, 14, 15]
```

Custom Reducers

We need two custom operations:

1. A digit histogram reducer that aggregates cell digits into per-unit histograms (via Σ)
2. A duplicate counter morphism that counts violations

```
function digit_histogram_reducer(source_values, relation, metadata)
    result = Dict{Any, Any}()
    for (unit_name, cell_indices) in relation
        counts = Dict(d => 0 for d in 0:NUM_DIGITS-1)
        for idx in cell_indices
            digit = get(source_values, idx, -1)
            if digit >= 0
                counts[digit] = get(counts, digit, 0) + 1
            end
        end
        result[unit_name] = counts
    end
    result
end
```

```
function count_duplicates(histogram)
    Dict(
        unit => sum(max(0, count - 1) for (_, count) in counts)
        for (unit, counts) in histogram
    )
end
```

```
count_duplicates (generic function with 1 method)
```

Building the Constraint Diagram

Now we construct the FunctorFlow diagram. Three Σ (left Kan) extensions aggregate cell digits into row, column, and block histograms. Three morphisms then count violations.

```
D = @functorflow SudokuConstraints begin
    CellDigits :: cell_digits
    RowUnits :: row_relation
    ColUnits :: col_relation
    BlockUnits :: block_relation
end

# Add Σ extensions for each constraint dimension
Σ(D, :CellDigits; along=:RowUnits, reducer=:row_hist, name=:row_histograms)
Σ(D, :CellDigits; along=:ColUnits, reducer=:col_hist, name=:col_histograms)
Σ(D, :CellDigits; along=:BlockUnits, reducer=:block_hist, name=:block_histograms)

# Add violation-counting morphisms
add_object!(D, :RowViolations; kind=:violation_map)
add_object!(D, :ColViolations; kind=:violation_map)
add_object!(D, :BlockViolations; kind=:violation_map)

add_morphism!(D, :row_dupes, :row_histograms, :RowViolations;
              implementation=count_duplicates)
add_morphism!(D, :col_dupes, :col_histograms, :ColViolations;
              implementation=count_duplicates)
add_morphism!(D, :block_dupes, :block_histograms, :BlockViolations;
              implementation=count_duplicates)

# Bind the custom reducer
bind_reducer!(D, :row_hist, digit_histogram_reducer)
bind_reducer!(D, :col_hist, digit_histogram_reducer)
```

```
bind_reducer!(D, :block_hist, digit_histogram_reducer)

println(D)
```

```
Diagram :SudokuConstraints ⟨10 objects, 3 morphisms, 3 Kan, 0 losses⟩
```

Generating Puzzles

We generate random valid 4×4 Sudoku solutions and mask some cells to create puzzles.

```
function random_solution(rng=Random.default_rng())
    base = [0 1 2 3; 2 3 0 1; 1 0 3 2; 3 2 1 0]
    # Permute digits
    perm = randperm(rng, NUM_DIGITS) .- 1
    board = [perm[d+1] for d in base]
    # Shuffle row bands
    if rand(rng) < 0.5
        board = board[[2,1,3,4], :]
    end
    if rand(rng) < 0.5
        board = board[[1,2,4,3], :]
    end
    if rand(rng) < 0.5
        board = board[[3,4,1,2], :]
    end
    # Shuffle column bands
    if rand(rng) < 0.5
        board = board[:, [2,1,3,4]]
    end
    if rand(rng) < 0.5
        board = board[:, [1,2,4,3]]
    end
    if rand(rng) < 0.5
        board = board[:, [3,4,1,2]]
    end
    vec(board')  # row-major flattening
end

function mask_puzzle(solution, num_givens; rng=Random.default_rng())
    indices = randperm(rng, NUM_CELLS)
    given_set = Set(indices[1:num_givens])
```

```
    [i in given_set ? solution[i] : -1 for i in 1:NUM_CELLS]
end

function format_grid(values)
    rows = [values[(i-1)*BOARD_SIZE+1 : i*BOARD_SIZE] for i in 1:BOARD_SIZE]
    join([join(string.(r), " ") for r in rows], "\n")
end

rng = Random.MersenneTwister(42)
solution = random_solution(rng)
puzzle = mask_puzzle(solution, 8; rng=rng)
println("Solution:")
println(format_grid(solution))
println("\nPuzzle (8 givens, -1 = blank):")
println(format_grid(puzzle))
```

```
Solution:
3 2 0 1
0 1 3 2
2 3 1 0
1 0 2 3

Puzzle (8 givens, -1 = blank):
-1 2 0 -1
0 1 3 2
2 -1 -1 -1
-1 -1 -1 3
```

Analyzing Constraint Violations

Now we run the diagram to check whether a board satisfies all Sudoku constraints. We feed cell digits and unit relations as inputs.

```
function analyze_board(board)
    rels = build_unit_relations()
    # Convert 1-indexed Julia board to 0-indexed cell mapping
    cell_digits = Dict(i-1 => board[i] for i in 1:NUM_CELLS)

    compiled = compile_to_callable(D)
    result = FunctorFlow.run(compiled, Dict(
        :CellDigits => cell_digits,
```

```
        :RowUnits => rels.rows,
        :ColUnits => rels.cols,
        :BlockUnits => rels.blocks
    ))

    row_v = result.values[:row_dupes]
    col_v = result.values[:col_dupes]
    block_v = result.values[:block_dupes]
    total = sum(values(row_v)) + sum(values(col_v)) + sum(values(block_v))

    (row_violations=row_v, col_violations=col_v, block_violations=block_v,
     total_duplicates=total, is_consistent=total == 0,
     row_histograms=result.values[:row_histograms],
     col_histograms=result.values[:col_histograms],
     block_histograms=result.values[:block_histograms])
end
```

```
analyze_board (generic function with 1 method)
```

Checking a valid solution

```
result_good = analyze_board(solution)
println("Valid solution analysis:")
println("  Row violations:   ", result_good.row_violations)
println("  Col violations:   ", result_good.col_violations)
println("  Block violations: ", result_good.block_violations)
println("  Total duplicates: ", result_good.total_duplicates)
println("  Is consistent:    ", result_good.is_consistent)
```

```
Valid solution analysis:
  Row violations:   Dict(:row_0 => 0, :row_1 => 0, :row_3 => 0, :row_2 => 0)
  Col violations:   Dict(:col_0 => 0, :col_2 => 0, :col_3 => 0, :col_1 => 0)
  Block violations: Dict(:block_2 => 0, :block_0 => 0, :block_1 => 0, :block_3 => 0)
  Total duplicates: 0
  Is consistent:    true
```

A valid solution has zero violations in every row, column, and block.

Checking an invalid board

```
# Create an intentionally invalid board: repeat digit 0 in row 0
bad_board = copy(solution)
bad_board[1] = bad_board[2]  # duplicate in row 0

println("Invalid board:")
println(format_grid(bad_board))

result_bad = analyze_board(bad_board)
println("\nInvalid board analysis:")
println("  Row violations:   ", result_bad.row_violations)
println("  Col violations:   ", result_bad.col_violations)
println("  Block violations: ", result_bad.block_violations)
println("  Total duplicates: ", result_bad.total_duplicates)
println("  Is consistent:    ", result_bad.is_consistent)
```

```
Invalid board:
2 2 0 1
0 1 3 2
2 3 1 0
1 0 2 3

Invalid board analysis:
  Row violations:   Dict(:row_0 => 1, :row_1 => 0, :row_3 => 0, :row_2 => 0)
  Col violations:   Dict(:col_0 => 1, :col_2 => 0, :col_3 => 0, :col_1 => 0)
  Block violations: Dict(:block_2 => 0, :block_0 => 1, :block_1 => 0, :block_3 => 0)
  Total duplicates: 3
  Is consistent:    false
```

The duplicate in row 0 is detected as a constraint violation.

Inspecting the Histograms

The intermediate Σ (left Kan) results are the digit histograms — the raw aggregation before violation counting.

```
println("Row histograms for valid solution:")
for (unit, hist) in sort(collect(result_good.row_histograms); by=first)
    digits = join(["$d⇒$c" for (d,c) in sort(collect(hist))], ", ")
    println("  $unit: {$digits}")
end
```

```
Row histograms for valid solution:
  row_0: {0=>1, 1=>1, 2=>1, 3=>1}
  row_1: {0=>1, 1=>1, 2=>1, 3=>1}
  row_2: {0=>1, 1=>1, 2=>1, 3=>1}
  row_3: {0=>1, 1=>1, 2=>1, 3=>1}
```

In a valid solution, every unit histogram has exactly one occurrence of each digit.

ACSet Representation

The constraint diagram can be converted to an ACSet for Catlab integration:

```
acs = to_acset(D)
println("ACSet representation:")
println("  Nodes: ", nparts(acs, :Node))
println("  Edges: ", nparts(acs, :Edge))
println("  Kan extensions: ", nparts(acs, :Kan))
```

```
ACSet representation:
  Nodes: 10
  Edges: 3
  Kan extensions: 3
```

Interpretation

This vignette demonstrates a key insight: constraint satisfaction is aggregation. Each Sudoku constraint (no repeats in a row/column/block) is expressed as:

1. Σ (left Kan extension): aggregate cell digits along the unit's incidence relation □ produces a digit histogram
2. Morphism: count violations in the histogram □ produces a scalar

The same pattern applies to any constraint satisfaction problem where: - There is a set of variables (cells) - There are structured groups of variables (rows, columns, blocks) - Each group must satisfy a local constraint (no duplicates)

The Kan extension provides the universal aggregation — it collects all relevant data for each constraint check. The morphism then applies the domain-specific test.

This categorical decomposition is modular: adding a new constraint type (e.g., diagonal Sudoku) requires only adding a new relation object and another Σ extension. The rest of the diagram is unchanged.