Constraint Folding

AI が観測から法則を発見し、その法則がそのまま動くシステムになる —— ai-desk の最深部の機構

ai-desk 方法論 — 沖井広行 · 2026-05

1 表面に見える顔 — 「分岐を data に置く」

第一印象では、Constraint Folding はただの「if/else を テーブルに置き換えるテクニック」に見える。

```
// よくある「分岐 → データ化」の見せ方
const TRANSITIONS = [
  { from: 'title',    input: 'start', to: 'playing'  },
  { from: 'playing',  input: 'pause', to: 'paused'   },
  // ...
];
function reduce(state, input) {
  return TRANSITIONS.find(t => t.from === state && t.input === input)?.to ?? state;
}
```

これは **入口** にすぎない。本当の力はその先にある。

2 本当の顔 — 「観測から法則を発見する基盤」

Constraint Folding の核心は、AI が観測データから世界 (テーブル) を推測し、それがそのまま動くこと。

> 人間が **意図と観測サンプル**を渡す
> → AI が **法則を抽出してテーブルを生成**する
> → そのテーブルが **そのまま動く state machine** になる

「コードを書く」というフェーズが消える。代わりに「**観測から法則を抽出する**」フェーズが生まれる。

これは Bible §6「鉱脈採掘パラダイム」と Constraint Folding が結合した時に現れる挙動。

3 ４フェーズ

1
観測
既存システムや業務ルールから (入力, 出力) のサンプルを集める

2
候補生成
LLM + Monte Carlo で「ありうる法則」を多数列挙

3
検証
各候補を全観測サンプルに通し、整合するもの以外を捨てる

4
構築
残った法則 = 制約テーブル = 即動く state machine

人間が触るのは１(観測を渡す) と４(動作確認) だけ。間の２と３は完全に AI。

4 実例 — 送料計算 50 サンプル → 100% 復元

レガシー送料ロジックのソースが失われた状態から、観測 50 件で完全復元。

50	100%	0
観測サンプル数 (入力 → 出力ペア)	復元精度 (全 50 件で整合)	人間が書いた if 文 (テーブルだけで実装)

具体: `examples/` ディレクトリに PoC コードあり。
50 件の (重さ, 距離, 区分, 送料) ペアから、AI が「閾値・基本値・加算ルール」のテーブルを推論。
復元されたテーブルを reduce 関数に通すだけで全観測が再現される。**ソースコードを読まずにシステムを再構築した。**

5 なぜ「畳み込み」だから推測が可能か

純粋な reducer 関数だから、AI が 機械的に候補を試して絞り込める。

- **同じ入力 → 同じ出力** (純粋性) → 観測は再現可能、検証も再現可能
- **状態と入力が data** → 全可能組合せが列挙可能 → 候補生成時に「考えられるすべての組合せ」を網羅できる
- **制約は filter の述語** → 候補を「サンプルを通すか」でふるい落とす操作が単純な filter
- **逆方向に走らせられる** → 「この出力にするにはどんな入力ありうるか」を逆引きで生成 → これが推測候補の幅を広げる

分岐ツリーだとこれらが原理的に不可能 (制御フローは逆走できない、組合せは爆発、検証は副作用込み)。
畳み込みだから AI が推論を回せる。

6 推測されたテーブル = 即動く state machine

推論結果がそのまま実行可能。「コード生成」フェーズが要らない。

```
// AI が推論で生成したテーブル（人間は書いてない）
const SHIPPING_RULES = [
  { weight_max: 1,  zone: 'kanto',    base: 300, per_kg: 0   },
  { weight_max: 5,  zone: 'kanto',    base: 500, per_kg: 100 },
  { weight_max: 1,  zone: 'kansai',   base: 350, per_kg: 0   },
  // ...
];

// この reducer は固定（汎用）。テーブルを変えるだけで挙動が変わる
function resolve(weight, zone) {
  const rule = SHIPPING_RULES.find(r => r.zone === zone && weight <= r.weight_max);
  return rule ? rule.base + (weight * rule.per_kg) : null;
}
```

AI が推論した瞬間に、すでに動いている。**「推論結果を人間がコードに書き写す」が消える。**

7 二段構造 — 列挙 (基本) + 推測 (応用)

基本: 列挙生成

人間が軸を定義 → cartesian product でテーブル展開

```
HANDS.flatMap(a =>
  HANDS.flatMap(b =>
    HANDS.map(c => ({a,b,c}))
  )
);  // 27 世界
```

既知ドメインの全網羅

応用: 推測生成

観測サンプル → AI が制約を逆推論 → テーブル自動生成

```
// 50 サンプルから
const rules = infer(observations);
// AI が SHIPPING_RULES を生成
// → そのまま reducer に渡せる
```

未知ドメインの法則発見

どちらもアウトプットは同じ「テーブル + reducer」。**同じ実行基盤を、人間が書く / AI が推測する、の二系統で使える。**

8 何が変わるか

「コードを書く」が「観測から法則を抽出する」に置き換わる。

従来の開発

- 仕様を読む / 書く
- 仕様を if/else に翻訳する
- 分岐を全網羅的にテストする
- 仕様変更で全分岐を見直す
- 動かしてバグ修正

CONSTRAINT FOLDING + 推測

- 動くサンプル / 観測を用意する
- AI が法則を推論 → テーブル生成
- テーブルがそのまま reducer で動く
- 仕様変更 = サンプル追加 → 再推論
- バグ = 観測との不整合として即検知

人間が「翻訳作業」をしなくなる。観測と動作確認だけ持つ。

9 適用領域

レガシー復元

ソースが失われた業務ロジックを観測から再構築。送料・税計算・割引判定など。

新規ドメイン発見

「こういう挙動が欲しい」サンプルから AI がルールを推測 → 即動く実装。

既存システム監査

運用中システムの観測から「実装が想定通りか」を逆検証。仕様と実装の差分検出。

業務ルール自動化

担当者の判断履歴 → ルール推論 → 自動化。「暗黙のノウハウ」をデータ化。

ゲームバランス調整

プレイログから「楽しい状態の組合せ」を抽出 → ルール調整。

制約ソルバとしての汎用利用

有限離散ドメイン全般。スケジューリング・配置・組合せ最適化。

10 ai-desk の最終形 — 「コード」概念の解体

究極的には「コードを書く」というステップが消える。

> **残るのは 3 つだけ:**
> ・人間の **意図 + 観測サンプル**
> ・AI の **法則推論**
> ・固定の **reducer 実行基盤**

「コードベース」が「**制約テーブルの集合**」になる。
「リファクタリング」が「**サンプル追加して再推論**」になる。
「バグ修正」が「**失敗観測を加えて再構築**」になる。

これは Bible §0.0 「複雑性は人の問題、隠匿は AI の問題」を最終的に解決する形。
人間は **意図と観測**、AI は **推論と構築**、機械は **純粋実行**。役割が完全に分離する。

観測 → 法則 → 実行

Constraint Folding は「データ化テクニック」じゃなく、
AI が法則を発見し、それがそのまま動く ための土台。

列挙は入口、**推測こそが本丸。**
この機構の上では、開発は「書く」ではなく「**抽出する**」になる。

github.com/AoyamaRito/ai-desk

沖井広行 · 蒼山りと