koba-e964 MasWag nullpo-head wasabiz zeptomoter

yabaitech.tokyo vol.3

2019.09.22@ 技術書典 7

本誌の PDF とソースコードのダウンロードはこちらから

https://bit.ly/ytt-2cu4p

目次

Schreier-Sims のアルゴリズムを Rust で実装した

koba-e964

1. はじめに

趣味で数学を勉強している kobae964 です。置換群の位数を求めるための Schreier-Sims のアルゴリズム [*1]というアルゴリズムを実装したので、 それの解説を書いていきたいと思います。

GAP [2] などの計算代数システムを使うと、 色々な計算をたちどころに行うことができて非常に便利です。 しかし、 その内部でどのような処理が実行されているのかは、 知らない方も多いのではないでしょうか？置換群の位数の計算もその一つのように思えます。 例えば $3 \times 3 \times 3$ ルービックキューブの可能な配置は有限群をなすので、その総数は、GAP を使用すれば簡単に計算できます。[1] しかし、そのために内部でどのような処理を必要とするのかは決して自明ではありません。 ルービックキューブの配置の総数は $43252003274489856000 \simeq 4.3 \cdot 10^{19}$[*2]であり、愚直に一つ一つの配置を見ていくやり方では決して全数検査を行うことはできません。 これの内部で実行されているのが Schreier-Sims のアルゴリズムです。n 点の置換群について、n についての多項式時間で位数を計算することができます。

この記事では、Schreier-Sims のアルゴリズムのアイディア、 実装を追っていけたらと

---

1 https://en.wikipedia.org/wiki/Schreier%E2%80%93Sims_algorithm

2 $43252003274489856000 = 2^{11} \times 3^7 \times 12! \times 8!/2$

思います。今回はアルゴリズムの基本方針についてです。なお、実装は GitHub のリポジトリ [3] に置いてあるので、興味のある方は読んでいただけたらと思います。

2. 前提知識

群論の初歩的な知識、およびグラフ理論の初歩的な知識を仮定する。また擬似コードも Rust 風に書くので、Rust の知識があることも望ましい。

- 群論 : (例えば [6])
- グラフ理論
  - 深さ優先探索 (例えば [8])
  - 幅優先探索 (例えば [7])
- Rust (例えば [10])

3. 記法

群の作用は右作用とする。つまり、x に対する g の作用を x^g と表記する。これは GAP における流儀と合わせている。よく使われる左作用の記法 $(g \cdot x)$ と逆であることに注意されたい。作用は $x^{g \cdot h} = (x^g)^h$ を満たす必要がある。これも左作用の公理 $((g \cdot h) \cdot x = g \cdot (h \cdot x))$ とは逆である。

4. 導入

計算群論において、Schreier-Sims のアルゴリズムは置換群の位数を計算したり、置換群にある要素が含まれているかを高速に判定したり ... などのクエリへの高速な応答を行うための前計算アルゴリズムである。今回は置換群 G の位数 $|G|$ を求めるところに焦点を当てて議論する。形式的に表記すると、対称群 S_n が集合 $\{0, 1, ..., n-1\}$ の上に作用するとしたとき、我々の興味は生成元の集合 $X = \{x_1, ..., x_k\} \subseteq S_n$ によって生成された群 $G = \langle X \rangle \subseteq S_n$ の位数を計算することである。これにより例えば以下のことが可能になる :

- $3\times3\times3$ ルービックキューブのありうる配置の総数を求める。キューブの面は合計 54 個あり、各面の中心は動かさないとしてよいので、合法的な操作は S_{48} の部分群をなすとみなすことができる。(もっと精密な議論をすると、例えばコーナー・キューブとエッジ・キューブが互いに移り変わることはないため $S_{24}\times S_{24}$ の部分群であるということも可能であるが、ここではこの事実は使用しない。)

5. 軌道安定化群定理

一般に部分群の位数や指数を求めるのは簡単なことではない。しかし安定化群[*3]に限って言えば指数を求めるのは比較的簡単である。それには以下の定理が重要である。

定理 1　群 G を S_n の部分群とし、集合 $\{0,1,\ldots,n-1\}$ の上の自然な作用を考える。

$$x\in\{0,\ldots,n-1\}$$

の軌道[*4]を

$$\mathrm{Orb}_G(x):=\left\{x^g \;\middle|\; g\in G\right\},$$

x の安定化群を

$$\mathrm{Stab}_G(x):=\left\{g\in G \;\middle|\; x^g=x\right\}$$

と表記することにする。このとき、

$$\left|\mathrm{Orb}_G(x)\right|\cdot\left|\mathrm{Stab}_G(x)\right| = |G|$$

が成り立つ。　◇

$\mathrm{Orb}_G(x)$ はグラフ理論的なアルゴリズム (深さ優先探索) で効率的に計算できるため、安定化群の大きささえわかれば $|G|$ がわかる。

3　https://ja.wikipedia.org/wiki/ 群作用 # 軌道と等方部分群

4　https://ja.wikipedia.org/wiki/ 群作用 # 軌道と等方部分群

6. 安定化群を用いた減少列の構成

軌道安定化群定理を用いて次々に安定化群を作っていくことを考える。$H_0 = G$ として、

$$H_1 = \mathrm{Stab}_{H_0}(u_0), H_2 = \mathrm{Stab}_{H_1}(u_1), \ldots, H_k = \mathrm{Stab}_{H_{k-1}}(u_{k-1}) = \{e\}$$

として、$G = H_0 \supseteq H_1 \supseteq H_2 \cdots \supseteq H_k = \{e\}$ という部分群の列を構成したとする。このとき

$$|G| = (H_0 : H_1)(H_1 : H_2) \cdots (H_{k-1} : H_k)$$

が成り立つ。各 $(H_i : H_{i+1}) = \left|\mathrm{Orb}_{H_i}(u_i)\right|$ が計算できれば $|G|$ も計算できる。このようにして順番に全ての点を固定して、各時点での軌道の大きさを求め、それら全てを掛け合わせることで $|G|$ を計算する、というのが Schreier-Sims のアルゴリズムによる位数計算の基本的なアイディアである。以上の議論において、各ステップで計算すべきものは二つある。

- G の生成元と x が与えられたとき、$\mathrm{Orb}_G(x)$
- G の生成元と x が与えられたとき、$\mathrm{Stab}_G(x)$ の生成元

$\mathrm{Orb}_G(x)$ は G の作用によって到達可能な点全体の集合であるため、幅優先探索か深さ優先探索で計算できる。$\mathrm{Stab}_G(x)$ の生成元は難しいが以下のアイディアによって計算できる。

- $\mathrm{Orb}_G(x)$ の計算中にグラフの探索によって $x^p = x^q$ である $p, q \in G$ が見つかったとする。この時、$x^{p^{-1} \cdot q} = x$ であるため、$p^{-1} \cdot q \in \mathrm{Stab}_G(x)$ である。
- 逆に、$\mathrm{Stab}_G(x)$ の元はすべてこのような元の積で表せる。(Schreier's Theorem, [9] の p.8)

これを認めれば $\mathrm{Stab}_G(x)$ の生成元も計算できる。実装には幅優先探索を用いた。以下に擬似コード ([9] の p.6) を与える。実装本体は GitHub リポジトリ[*5]にある。

5 https://github.com/koba-e964/rust-schreier-sims/blob/754d0ad1/src/main.rs#L76-L110

```
fn orbit_stabilizer(n: 要素数 , gen: G の生成元のリスト , v: 固定する点) -> (軌道 , 安定化群の生成元) {
    que: キュー
    stabilizer_gen: 安定化群の生成元のリスト
    (v, e) を que に積む (e は単位元)
    stabilizer_gen <- []
    for (y, g) in que {
        if y に訪れたことがある {
            p(v) = y という情報が記録されていれば、 p^{-1} * g は v を固定する。
            gen.push(p^{-1} * g);
            continue;
        }
        y に訪れたことにし、g(v) = y という情報を記録する
        for x in gen {
            (x(y), g * x) を que に積む
        }
    }
    return (orb, stabilizer_gen)
}
```

7. Schreier-Sims のアルゴリズム (愚直)

0 から $n-1$ までの点を順番に固定して、徐々に群を小さくするというのが基本方針である。[*6]上の数式でいうと $u_0 = 0, u_1 = 1, \ldots, u_{n-1} = n-1$ である。

```
fn schreier_sims(n: 点数 , x: 生成元の集合) -> 位数 {
    gen := x;
    ord := 1;
    for i in 0..n {
        (orb, stab) := orbit_stabilizer(n, gen, i);
        ord *= |orb|;
        gen = stab;
    }
    return ord;
}
```

6 ソースコード : https://github.com/koba-e964/rust-schreier-sims/blob/754d0ad1/src/main.rs#L126-L151

これの計算量はどうなるだろうか？各ステップで、生成元の個数は最悪の場合 (軌道の大きさ) 倍になる。i を固定するときの軌道の大きさは最大で $n - i$ であるため、これらの積は最悪 $O(n!)$ 程度である。当初の目標は $n = 48$ の場合に群の位数を求めることだったため、これでは到底間に合わない。多項式時間バージョンの Schreier-Sims のアルゴリズムはより賢い工夫をしている。それの解説は次節以降に譲る。

8. Schreier-Sims のアルゴリズム (多項式時間)

前節のアルゴリズムを高速化し、計算量が多項式に収まるようにする。この改善は Sims [4] による。そもそも問題は、固定化群の生成元が多くなりすぎることだったので、そうならないような工夫をすればよい。

定義 2 (BSGS)　正確な定義は [9] に譲るとして、感覚的な概念を説明する。Base and strong generating set (BSGS) とは、以下の列 2 個のペア $\langle \beta, S \rangle$ である。

- β_i $(1 \leq i \leq l)$: 頂点の列。S を"だんだん固定"していく列。
- S_i $(1 \leq i \leq l)$: 生成元の集合の列。$S_1 \supseteq S_2 \supseteq \cdots S_l$ を満たす必要がある。また、$\langle S_i \rangle = \mathrm{Stab}_G(\{\beta_1, \ldots, \beta_i\})$ を満たさなければならない。

◇

定義 3 (Transversal)　$x \in X$ とする。各 $u \in \mathrm{Orb}_G(x)$ に対して $\alpha_u \in G$ であって $x^{\alpha_u} = u$ を満たすものを集めた集合 $\{\alpha_u \mid u \in \mathrm{Orb}_G(x)\}$ を transversal と呼ぶ。 ◇

上で固定化部分群をとったときは生成元が膨らんでいったが、BSGS についていえば生成元はだんだん小さくなっていく。(というよりも、だんだん小さくなるように余計な元を足しておくのである。) 一般の生成元が BSGS であるわけはないため、BSGS を得るためのアルゴリズムが必要である。修正された Schreier-Sims のアルゴリズムは、与えられた生成元の集合を、BSGS になるまで大きくしていく、というアイディアに基づいている。擬似コードは以下である。

```
fn orbit_transversal_stabilizer(n: 要素数 , gen: G の生成元のリスト , v: 固定する点) -
> (軌道と transversal の組 , 安定化群の生成元) {
    /* orbit_stabilizer と同様。軌道の計算と同時に transversal の計算もできることに注意。
```

```
*/
}
fn get_transversal(n: usize, orbit_transversal: 軌道と transversal の組) -> tr
ansversal の配列表現 {
    for a in 0..n {
        ans[a] = (a が軌道に含まれていれば x^g = a となる g を、そうでなければ None)
    }
    ans
}
// g を beta_transversal で示される固定化部分群の元の積として表そうとする。
// どうしても残ってしまう余りの部分も返す。
pub fn strip(
    g: 置換,
    beta_transversals: &[(usize, Transversal)]
) -> (Vec<置換>, 余りの置換) {
    h <- g;
    us <- [];
    for &(beta, ref transversal) in beta_transversals {
        let moved_to = h[beta];
        match transversal[moved_to] {
            ダミー => break,
            repr => { // repr は beta^{repr} = moved_to を満たす。
                h <- h * repr^{-1}
                us.push(repr.clone());
            }
        }
    }
    (us, h) // g = h * us[0] * ... * us[us.len() - 1] を満たす
}
// 与えられた (beta の列, s) が BSGS かどうか調べ、そうでない場合は何を追加すべきかを返す。
fn schreier_sims(
    n: usize,
    beta_transversals: &[(usize, Transversal)],
    s: 置換の列,
) -> Result<(), (Vec<置換>, 余りの置換)> {
    if beta_transversals.is_empty() {
        if s.is_empty() {
            return Ok(());
        }
        return Err((vec![], s[0].clone()));
    }
```

```
    // The first fixed point
    let beta0 = beta_transversals[0].0;
    let intersection = (s の中で、beta0 を保つ置換全体);
    schreier_sims(n, &beta_transversals[1..], &intersection)?; // 再帰
    // y はナイーブな方法で作った固定化群の生成元
    let (_, y) = orbit_stabilizer(n, s, beta0);
    for y in y {
        let (us, 余り) = strip(&y, &beta_transversals[1..]);
        if 余り が e でない {
            異常終了 , (us, 余り) を返す
        }
    }
    正常終了
}
fn incrementally_build_bsgs(
    n: usize,
    initial_beta: &[頂点],
    initial_s: &[置換],
) -> (Vec<(頂点 , Transversal)>, Vec< 置換 >) {
    let mut beta_transversals = vec![ダミーデータ ; initial_beta.len()];
    let mut s = initial_s.to_vec();
    let mut used = vec![false; n];
    let dummy_transversal = vec![]; // dummy transversals
    loop {
        {
            // transversal を計算する。後の schreier_sims 用。
            let mut cur_s = s.clone();
            // beta を固定する固定化部分群のようなものを雑にとっていく。
            // (beta_transversal, s) が BSGS であれば、これは実際に固定化部分群になる。
            for (beta, transversal_ref) in beta_transversals {
                let (orbit_transversal, _) =
                    orbit_transversal_stabilizer(n, &cur_s, beta);
                let transversal = get_transversal(n, orbit_transversal);
                *transversal_ref = transversal;
                cur_s = (cur_s の中で beta を固定する置換全体);
                used[beta] = true;
            }
        }

        // スライド内の Y をインクリメンタルに計算し、問題ないか調べる。
        match schreier_sims(n, &beta_transversals, &s) {
```

```
            正常終了 => break,
            (_, h) を返して異常終了 => {
                s に h を追加;
                // beta に含まれ、h で固定化されない頂点はあるか？
                let mut moved = (h で固定化されない頂点の列);
                if moved と beta の共通部分がない かつ moved が空でない {
                    // h で固定化されない頂点は beta の中にはない。
                    // その中から一つランダムに選ぶ
                    let point = (moved からランダムにとる);
                    beta_transversals に (point, dummy_transversal) を追加;
                }
            }
        }
    }
    (beta_transversals, s)
}
```

実装は schreier::incrementally_build_bsgs に与えた。[*7]

9. 群の位数の計算

ここまでくればそこまで難しくない。擬似コードは以下である。

```
// n: 頂点数, gen: 生成元
fn order(n: usize, gen: &[置換]) -> 整数 {
    let mut order = 1;
    let (bsgs, _) = incrementally_build_bsgs(n, &[], gen);
    for (_, transversal) in bsgs {
        let u = (traversal から計算した (H_i:H_{i+1}) の値);
        order *= u;
    }
    order
}
```

7 ソースコード : https://github.com/koba-e964/rust-schreier-sims/blob/4e0be968/src/schreier.rs#L59-L116

実装は schreier::order に与えた。[*8]

10. 謝辞

この記事は以下を参考にして書かれました。

- [9]: このスライドを主に利用して実装しました。

参考文献

[1] 藤本 光史 . ルービックキューブと Gap. , pages 1–5, 2006.

[2] GAP Group. *GAP System for Computational Discrete Algebra*. https://www.gap-system.org/, 2019.

[3] koba-e964. . https://github.com/koba-e964/rust-schreier-sims, 2019.

[4] Murray, Scott H and O'Brien, Eamonn A. Selecting base points for the Schreier-Sims algorithm for matrix groups. *Journal of Symbolic Computation*, 19(6), pages 577–584, 1995.

[5] Murray, Scott H. *The Schreier-Sims algorithm*. http://www.maths.usyd.edu.au/u/murray/research/essay.pdf, 2003.

[6] . 群論 . https://ja.wikipedia.org/wiki/%E7%BE%A4%E8%AB%96, 2019.

[7] . 幅優先探索 . https://ja.wikipedia.org/wiki/ 幅優先探索 , 2019.

[8] . 深さ優先探索 . https://ja.wikipedia.org/wiki/ 深さ優先探索 , 2019.

[9] Holt, Derek. *The Schreier-Sims algorithm for finite permutation groups*. https://blogs.cs.st-andrews.ac.uk/codima/files/2015/11/CoDiMa2015_Holt.pdf, 2015.

8 ソースコード : https://github.com/koba-e964/rust-schreier-sims/blob/4e0be968/src/schreier.rs#L118-L132

[10] The Rust team. *Rust Programming Language*. `https://www.rust-lang.org`, 2019.

はじめての AI

—AI は AI でも Automata Inference の方だがなぁぁーっ !—

MasWag

1. はじめに

俺の名前は兼江 直次 (かなえ なおつぐ)。
埼玉県の川越市に在住の 17 歳の高校生だ。
今日から夏休み !
俺は億万長者になる為に AI の勉強に行く事にした。

さて、 どこに AI を勉強しに行くか ...
やはり基本は山だろう !
山に行く !

俺は山にやって来た。
すると ...

なぜか山で水着の女の子が現れた。
女の子 :「AI は好きですか ?

直次 :「好き。
女の子 :「本当 ⁉ 嬉しい ...
女の子は喜びに大きく目を ...
見開いた !
女の子 :「AI は AI でも Automata Inference の方だがなぁぁーっ !

本章では正規言語 [1] を神託を用いて正確に学習する手法 (automata learning, automata inference) について説明します。本章において、形式言語理論についての知識は仮定しませんが、素朴集合論や基本的なグラフ理論の知識や記号などは適宜使用します。Automata learning では deterministic finite automaton(決定的有限オートマトン、DFA) を学習する L* アルゴリズム [2] が良く知られています。有限の文字集合を Σ したとき、L* アルゴリズムはある文字列 $w \in \Sigma^*$ が学習したい言語 $L \subseteq \Sigma^*$ に含まれているか ($w \in L$) を

質問する**所属性質問** (membership query) と、学習した言語 $L' \subseteq \Sigma^*$ が学習したい言語 $L \subseteq \Sigma^*$ と等しいかどうかを質問する**等価性質問** (equivalence query) の二種類の質問を繰り返すことで効率良く、学習したい言語を最小の状態数を持つオートマトンで学習します。本章では L* アルゴリズムに加えて、等価性質問を用いない ID アルゴリズム [3] と、DFA の代わりに non-deterministic finite automaton(非決定的有限オートマトン、NFA) を学習する NL* アルゴリズム [4] も説明します。

1.1. 本章で扱わない項目

本章では DFA 及び NFA を学習する手法について説明しますが、それ以外の様々な種類のオートマトンに対しても、学習アルゴリズムが研究されています。例えば重み付き有限オーマトン (weighted finite automaton, WFA)[5] については Balle & Mohri による手法 [6]、シンボリックオートマトン (Symbolic automaton)[7] については Drews 等による手法 [8]、レジスタオートマトン (Register automaton)[9] については Cassel 等による手法 [10] などが知られています。オートマトン学習の実装としては、LearnLib[11]、libalf [12]、AIDE[13] などが知られていますが、2019 年夏現在では LearnLib が最も良くメンテナンスされているようです。オートマトン学習の応用として、例えばモデル検査と組み合わせた Black-box checking(Learning-based testing とも)[14] などが注目されています。

1.2. 本章の構成

本章の構成は以下のようになります。まず 2 節で形式言語やオートマトンの必要な定義を与えます。本節では必要最低限の内容のみを扱います。より詳細は [1] などを参照してください。3 節では Angluin の ID アルゴリズムの紹介を行います。4 節では Angluin の L* アルゴリズムについての紹介を行います。5 節 DFA の代わりに NFA を学習する NL* アルゴリズムについての説明を行います。

1.3. 記法

本章では実数の集合を $\mathbb{R}$、自然数の集合を $\mathbb{N}$ を用いて表わします。また、真理値を + と - を用いて表わします。数式中で変数及び単なる文字としてラテン文字が表われますが、原則として変数として用いるときには a の様に斜体で、文字として用いるときは a のように立体で表記します。

謝辞　本章の作成にあたり、川越が生んだ鬼才、高橋邦子先生に感謝します。

2. 準備

魔王：「駄目じゃないか！そんな変なもの食べちゃ！
猫：「俺に有限文字列を食えと言ったのはエミリーだ。
猫：「そして俺は正規言語の味に目覚めた。
魔王：「駄目！絶対、 駄目！
魔王：「猫は！
魔王：「キャットフード食べないと！
魔王：「駄目でしょおおおおおおおおおおおお！
猫：「俺が正しいか貴様が正しいか ...
猫：「決着を付ける必要があるようだな！
魔王：「望むところだぁぁぁぁぁぁぁぁぁぁぁぁぁぁ！
猫：「では行くぞおおおおおおおおおおおお！
猫：「決戦のバトル・フィールドへ！

本章では有限集合 Σ を**文字**の集合として用います。例えば電話番号の様な数字の列を考える場合、Σ としてアラビア数字の集合 $\{0,1,2,3,4,5,6,7,8,9\}$ を用います。文字の有限列を**文字列**と定義します。文字列 w, w' を結合するときには演算子 $\cdot$ を用いて $w \cdot w'$ と表記します。例えば文字の集合 Σ が決戦のバトル・フィールド、つまり $\Sigma = \{\mathrm{K},\mathrm{B},\mathrm{F}\}$ のとき、KBF や BBKKBKK などは文字列となります。これらを結合した $\mathrm{KBF} \cdot \mathrm{BBKKBKK}$、つまり KBFBBKKBKK も文字列です。また、長さ 0 の特別な文字列として ε を用います。任意の文字列 w について ε は $w \cdot \varepsilon = \varepsilon \cdot w = w$ の性質が成り立ちます。以後長さ n の文字列 $a_1 a_2 \cdots a_n$ の集合を Σ^n、有限長文字列の集合を $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^n$ と書きます。また、文字の集合として以後主に $\Sigma = \{\mathrm{K},\mathrm{B},\mathrm{F}\}$ を用います。

2.1. 形式言語とオートマトン

文字列の集合を**言語**と言います。例えば文字の集合 Σ が $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$ のときに Σ^* の部分集合である、3 桁の電話番号の集合やフリーダイアルの電話番号の集合などは言語の例です。偶数の集合や回文となっている文字列の集合も言語の例です。

非決定的有限オートマトン (nondeterministic finite automata, NFA) は、様々な言語の中

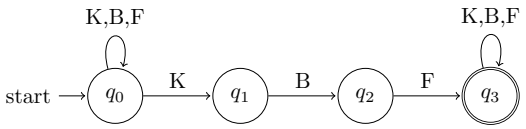


図 1　NFA の例 $\mathscr{A}$。"KBF"という部分列を含む文字列を受理する。

でも**正規言語**と呼ばれる言語を表現する方法の中の一つです。NFA は例えば図 1 にある、$\mathscr{A}$のような状態遷移図を用いて表されます。NFA $\mathscr{A}$が表現する言語を $L(\mathscr{A})$ と表記することとし、文字列 $w \in \Sigma^*$ が $L(\mathscr{A})$ に含まれるとき、$\mathscr{A}$が w を**受理する**と言います。図 1 にある NFA $\mathscr{A}$は、文字集合 $\{\mathrm{K,B,F}\}$ の上で定義された NFA で、「KBF という部分列が存在する文字列」を受理します。この NFA には状態が q_0, q_1, q_2, q_3 の 4 つありますがその中に特別な役割をする状態が二種類あります。一つ目は**初期状態**と呼ばれる状態です。本章では初期状態は"start"と書かれた矢印で外から指すことで表すこととします。今回の初期状態は q_0 のみですが、複数存在しても構いません。二つ目は**受理状態**と呼ばれる状態です。本章では受理状態は二重丸で表すこととします。今回の受理状態は q_3 ですが、こちらも複数存在しても構いません。

例として $\mathscr{A}$が文字列 $w = \mathrm{KKBF}$ を受理する様子を見ていきます。NFA ではまず始めに初期状態 q_0 にいるところから始めて、一文字ずつ文字を読んでいき、それに従って次の状態に進んでいきます。まず始めに一文字目の K を読みます。このとき、q_0 が始点で K のラベルが付いている辺の終点は q_0 と q_1 の二つがあるので、次の状態は q_0 及び q_1 となります。同様に二文字目の K を読むと、q_0 が始点で K のラベルが付いている辺の終点は q_0 及び q_1 で、q_1 が始点で K のラベルが付いている辺は存在しないので、次の状態も q_0 と q_1 となります。更に三文字目の B を読むと、q_0 が始点で B のラベルが付いている辺の終点は q_0 で、q_1 が始点で B のラベルが付いている辺の終点は q_2 なので、次の状態は q_0 と q_2 となります。最後に四文字目の F を読むと、q_0 が始点で F のラベルが付いている辺の終点は q_0 で、q_2 が始点で F のラベルが付いている辺の終点は q_3 なので、最後の状態は q_0 と q_3 となります。全ての文字を読み込んだ後で、現在居る状態に受理状態が存在するかどうかを調べます。今回の最後にいる状態のうち、q_3 は二重丸の付いている受理状態なので、NFA $\mathscr{A}$は $w = \mathrm{KKBF}$ を受理する、ということがわかります。同様の手順を踏むことで、任意の文字列 $w' \in \{\mathrm{K,B,F}\}^*$ について $\mathscr{A}$が w' を受理するかどうかを調べることができます。

NFA の形式的な定義は以下の様になります。

定義 1 (非決定的有限オートマトン)　文字集合 Σ 上の非決定的有限オートマトン $\mathscr{A}$ は 5 つ組 $\left(\Sigma, Q, Q_0, Q_F, \Delta\right)$ である。但し Σ は有限の文字集合、Q は有限の状態集合、$Q_0 \subseteq Q$ は初期状態、$Q_F \subseteq Q$ は受理状態の集合、$\Delta \subseteq Q \times \Sigma \times Q$ は状態遷移関係である。文字列 $w = a_1 a_2 \cdots a$ 対して、ある状態列 $q_0, q_1, \ldots, q_n$ で、$q_0 \in Q_0$、$\left(q_i, a_i, q_{i+1}\right) \in \Delta$、$q_n \in F$ が成り立つものが存在するとき、$\mathscr{A}$ は w を**受理する**という。また、$\mathscr{A}$ が受理する文字列の集合を $\mathscr{A}$ の**受理言語**といい、$L(\mathscr{A})$ と表記する。また、NFA $\mathscr{A}$ が言語 $L(\mathscr{A})$ を**認識する**、という。NFA によって受理される言語を**正規言語** (regular language) という。 ◇

NFA $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, \Delta\right)$ について、Q_0 が単集合で、各 $q \in Q$、$a \in \Sigma$ について $\left(q, a, q'\right)$ を充たす $q' \in Q$ が一意に定まるとき、$\mathscr{A}$ が**決定的である**といいます。決定的である有限オートマトンを**決定的有限オートマトン** (deterministic finite automaton, DFA) と言います。本章では省きますが、任意の非決定的有限オートマトンに $\mathscr{A}$ について、決定的有限オートマトン $\mathscr{A}'$ で $L(\mathscr{A}) = L\left(\mathscr{A}'\right)$ を充たすものが存在することが知られています。

2.2. Myhill-Nerode の定理

正規言語には単に NFA や DFA で表現できるという以外にも様々な面で「有限らしさ」があります。ここでは「有限らしさ」の特徴付けの一つである Myhill-Nerode の定理を見ていきます。

定義 2 (Myhill-Nerode 同値関係)　有限文字集合 Σ 上の言語 L について、Myhill-Nerode **同値関係** $R_L \subseteq \Sigma^* \times \Sigma^*$ を、次のように定義する。文字列 $w, w' \in \Sigma^*$ について wR_Lw' が成り立つのは以下のときである。 ◇

$$\forall w'' \in \Sigma^*. w \cdot w'' \in L \Leftrightarrow w' \cdot w'' \in L$$

定理 3 (Myhill-Nerode の定理)　有限文字集合 Σ 上の言語 L について、L が正規言語であることの必要十分条件は、商集合 Σ^*/R_L が有限集合となることである。 ◇

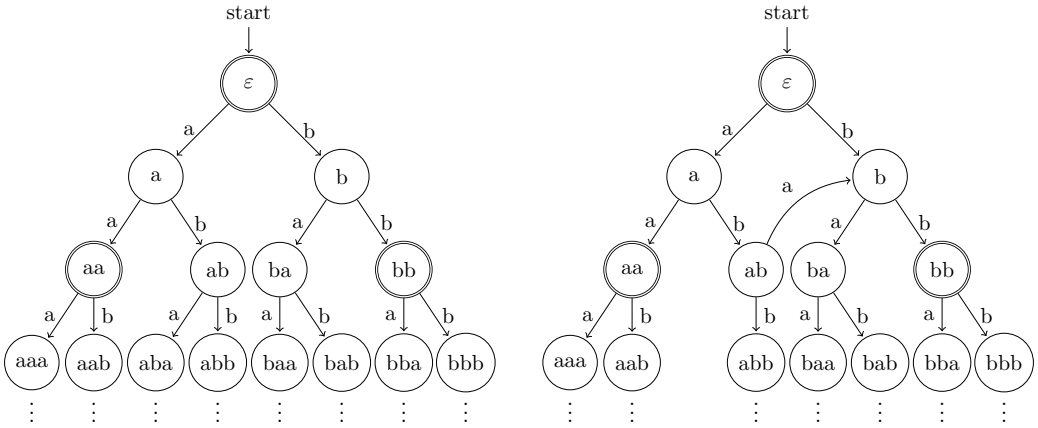


図 2　a, b の個数が共に偶数の文字列の言語を認識する無限状態のオートマトンの例 (右) 及び 状態 a と aba を纏めた無限状態のオートマトン (左)

文字列 w, w' と Myhill-Nerode 同値関係 R_L について、wR_Lw' が成り立つということの直観は、「これから与えられる未知の文字列 $w'' \in \Sigma^*$ について $w \cdot w'' \in L$ が成り立つかどうかを調べたいときに、今まで読んだ文字が w であるか w' であるかは気にしなくて良い」ということになります。このことについて木を使ってより詳しく見ていきます。文字集合 $\Sigma = \{\mathrm{a},\mathrm{b}\}$ 上の言語 L として、「a と b の出現回数が共に偶数回」というものを考えます。この言語を受理する (無限状態数の) オートマトンとして図 2 の木を考えます。この木では各文字列 $w \in \Sigma^*$ について一つの状態が割り当てられ、L に含まれる文字列と対応する状態が受理状態となります。ここで今回、例えば $\mathrm{b}R_L\mathrm{aba}$ が成り立ちますが、これは木の上の性質としてみると、b を根としたときの部分木と、aba を根としたときの部分木が等しいということになります。従って図 3 の (無限状態の) オートマトンのように、状態 b と状態 aba を一つにまとめても、受理言語は変わらないと言えます。商集合 Σ^*/R_L が有限集合、つまり右側のオートマトンのように状態を纏める操作を行うと最終的に有限状態のオートマトンになるとき、確かに $L(\mathscr{A}) = L$ が成り立つような、状態数 $\left|\Sigma^*/R_L\right|$ の DFA$\mathscr{A}$ を構成できるので L は正規言語になります。また、L が正規言語である、つまり $L(\mathscr{A}) = L$ が成り立つような DFA$\mathscr{A}$ が存在するとき、文字列 w, w' について $\delta\left(q_0, w\right) = \delta\left(q_0, w'\right)$ が成り立つならば wR_Lw' も成り立つので Σ^*/R_L は有限集合となります。以上が Myhill-Nerode の定理が成り立つ簡単な理由です。さらに、正規言語 L と DFA$\mathscr{A}$ について、状態数が $\left|\Sigma^*/R_L\right|$ より少ない場合 $\delta\left(q_0, w\right) = \delta\left(q_0, w'\right)$ であって wR_Lw' が成り立たないような文字列 $w, w' \in \Sigma^*$ が存在するので、次の系が成り立ち

ます。

系 4 (状態数最小の DFA)　正規言語 L について、状態数 $\left|L/R_L\right|$ で L を認識する *DFA* が存在し、また、状態数 L/R_L 未満の任意の *DFA* は L を認識しない。 ◇

3. ID アルゴリズム

絵美理：「Angluin の Automata Inference は嫌いですか ... ？
遼：「ソンナコトナイ！
遼：「オレ、 Angluin、 ダイスキ！

しかし初めて話す言葉がこれとは ...

絵美理：「嬉しい ...
絵美理は喜びに大きく目を ...

見開いた！
絵美理：「Angluin は Angluin でも ID アルゴリズム の方だがなぁぁぁーっ！

前節で定義した正規言語を NFA や DFA の形式で人間が書くことも当然できますが、これ以降本章では未知の正規言語を学習するという問題を主に考えます。 機械学習の教師付き二値分類問題では「正例と負例からなる訓練用データを与えて、 訓練用以外のデータについても高い確率で正しく分類する」 という問題設定が良く用いられますが、 ここでは exact learning と呼ばれる学習を考えます。Exact learning では、未知の正規言語を近似する正規言語を学習するのではなく、 未知の正規言語と完全に等しい正規言語を表現する NFA や DFA を学習します。

本節では学習対象の正規言語を表現する最小 DFA を学習する、ID アルゴリズム [3] を説明します。ID アルゴリズムでは訓練用データの代わりに、正規言語 L の live-complete な文字列集合 P と、 文字列 w が L に含まれるかを返答する神託を用いて学習を行います。

定義 5 (live-complete)　正規言語 $L \subseteq \Sigma^*$ について、文字列集合 $P \subseteq \Sigma^*$ が以下を充たすとき、P が live-complete であるという。但し $[w]_{R_L}$ は元 $w \in \Sigma^*$ の、Myhill-Nerode 同値関係 $R_L \subseteq \Sigma^* \times \Sigma^*$ 上の同値類である。

$$\left\{ [w]_{R_L} \;\middle|\; w \in P \right\} = \Sigma^*/R_L$$

言い替えると、L を認識する最小 DFA $\left(\Sigma, Q, \{q_0\}, Q_F, \Delta\right)$ の各状態 $q \in Q$ について、ある文字列 $w \in P$ が存在して、初期状態 q_0 から w を与えることで q に到達可能なとき P が live-complete であるという。 ◇

ID アルゴリズムの入力と出力は以下の様になります。 現実的にこれらの入力をどうやって生成するかは本章では扱いませんが、例えば文字列集合 P を十分大きく取ることで P が live-complete になると期待することができます。所属性神託については、例えば特定の正規言語に含まれるかどうかを判定することはわかっているが、内部状態などのわからないプログラムを用いることができます。

Exact learning (ID アルゴリズム)

入力 : 学習対象の正規言語を L としたときに、

- L の live-complete な有限の文字列集合 $P \subseteq \Sigma^*$
- $w \in \Sigma^*$ を与えて、$w \in L$ であるかどうかを返す神託 (所属性神託)

出力 : $L(\mathscr{A}) = L$ を充たす最小 DFA

本節では例として $L = \{a \in \{\mathrm{K,B,F}\} \mid \text{KBF を部分文字列として含む}\}$ を用います。例えば図 1 の DFA がこの言語を認識します。

3.1. 観察表

ID アルゴリズムでは**観察表** (observation table) と呼ばれる表 T に、各質問の結果を随時記入していくことで学習を進めていきます。 観察表は図 3 にある様に行、 列が共に Σ 上の文字列で添字付けされている表です。 観察表の各セルには所属性質問の結果が記入さ

	ε	F	BF
ε	−	−	−
K	−	−	+
KB	−	+	−
KBF	+	+	+
B	−	−	−
F	−	−	−
KK	−	−	+
KF	−	−	−
KBK	−	−	+
KBB	−	−	−
KBFK	+	+	+
KBFB	+	+	+
KBFF	+	+	+

図 3　観察表の例。$P = \{\varepsilon, \mathrm{K}, \mathrm{KB}, \mathrm{KBF}\}$, $P' = \{\mathrm{B}, \mathrm{F}, \mathrm{KK}, \mathrm{KF}, \mathrm{KBK}, \mathrm{KBB}, \mathrm{KBFK}, \mathrm{KBFB}, \mathrm{KBFF}\}$, $S = \{\varepsilon, \mathrm{F}, \mathrm{BF}\}$

れます。各セル $T[w, w']$ には文字列 $w \cdot w'$ の所属性質問の結果が記入されます。また、観察表には水平線より上側と下側の区別があります。水平線より上側の行の添字集合を $P \subseteq \Sigma^*$, 下側の添字集合を $P' \subseteq \Sigma^*$, また、列の添字集合を $S \subseteq \Sigma^*$ と呼びます。

観察表が以下の条件を充たすとき、その観察表を**一貫している** (consistent) と言います : 各 $w, w' \in P$ について、$T[w, \text{-}] = T[w', \text{-}]$ を充たすとき、任意の $a \in \Sigma$ について、$T[w \cdot a, \text{-}] = T[w' \cdot a, \text{-}]$ が成り立つ。

ID アルゴリズムにおいて、下側の添字集合 P' は $P' = \{w \cdot a \mid w \in P, a \in \Sigma, w \cdot a \notin P\}$ とします。

3.2. ID アルゴリズム

今回は $P = \{\varepsilon, \mathrm{K}, \mathrm{KB}, \mathrm{KBF}\}$ とします。このとき、$P' = \{w \cdot a \mid w \in P, a \in \Sigma, w \cdot a \notin P\}$ なので、$P' = \{\mathrm{B}, \mathrm{F}, \mathrm{KK}, \mathrm{KF}, \mathrm{KBK}, \mathrm{KBB}, \mathrm{KBFK}, \mathrm{KBFB}, \mathrm{KBFF}\}$ となります。最初は $S = \{\varepsilon\}$ となります。ID アルゴリズムでは図 4 の左の観察表から始めます。最初に空欄になっているセルを所属性質問を用いて埋めます (図 4 右)。

次に図 4 右の観察表が一貫しているかを判定します。今回は $\varepsilon, \mathrm{KB} \in P$ について、$T[\varepsilon, \text{-}] = [\mathrm{KB}, \text{-}]$ が成り立つ一方で $T[\mathrm{F}, \varepsilon] \neq T[\mathrm{KBF}, \varepsilon]$ なので一貫していません。そ

	ε
ε	
K	
KB	
KBF	
B	
F	
KK	
KF	
KBK	
KBB	
KBFK	
KBFB	
KBFF	

	ε
ε	-
K	-
KB	-
KBF	+
B	-
F	-
KK	-
KF	-
KBK	-
KBB	-
KBFK	+
KBFB	+
KBFF	+

図 4　一番始めの観察表 (左) と所属性神託を使って内容を埋めた観察表 (右)

こで閉じていない原因の列 F を S に追加して、観察表の空欄を埋めます (図 5)。

次に図 5 の観察表が一貫しているかを判定します。今回は $\varepsilon, \mathrm{K} \in P$ について、$T[\varepsilon, \text{-}] = T[\mathrm{K}, \text{-}]$ が成り立つ一方で $T[\mathrm{B}, \mathrm{F}] \neq T[\mathrm{KB}, \mathrm{F}]$ なので一貫していません。そこで閉じていない原因の列 F を BF に追加して、観察表の空欄を埋めます (図 3)。

今回は観察表が一貫しています。観察表が一貫している場合、次のルールに従って観察表から DFA を生成することができます。但し同値関係 $\approx_T \subseteq \left(P \cup P'\right) \times \left(P \cup P'\right)$ を $w \approx_T w' \Leftrightarrow T[w, \text{-}] = T[w', \text{-}]$ と定義します。

- 各 $[w]_{\approx_T} \in P/\approx_T$ に対して、DFA の状態 q_w を生成する
- DFA の初期状態は $q_{[\varepsilon]_{\approx_T}}$ とする
- DFA の受理状態は $T[w, \varepsilon] = +$ となる状態 q_w とする
- DFA の遷移関数 $\Delta\left(q_w, a\right)$ は、$w \cdot a \in P$ の場合 $\Delta\left(q_w, a\right) = q_{w \cdot a}$ とし、そうでない場合 $w \cdot a \approx_T w'$ を充たす $w' \in P$ について、$\Delta\left(q_w, a\right) = q_{w'}$ とする

例えば図 3 の観察表からは図 1 の DFA が生成されます。

今回の学習はここで終了となります。S に文字列を追加する操作の回数は高々 $|P|$ 回で済むので ID アルゴリズムは停止しますし、P が live-complete なので $L(\mathscr{A}) = L$ が成り

	ε	F
ε	−	−
K	−	−
KB	−	+
KBF	+	+
B	−	−
F	−	−
KK	−	−
KF	−	−
KBK	−	−
KBB	−	−
KBFK	+	+
KBFB	+	+
KBFF	+	+

図 5　$S = \{\varepsilon, \mathrm{F}\}$ のときの観察表

立ちます。さらに観察表 T が一貫しているとき、$\approx_T$ が Myhill-Nerode 同値関係となるので、$L(\mathscr{A})$ は L を認識する最小の DFA となります。

4. L* アルゴリズム

突然、 指名手配中の殺人鬼が現れた！
殺人鬼：「ぶっぽるぎゃるぴるぎゃっぽっぱぁぁぁぁぁーっ！
殺人鬼は奇声を上げて ...

Angluin 1987 年 L* を振り上げた！
そして殺人鬼はジャンプして思いっきりそれを ...

振り下ろした！

前節で説明した ID アルゴリズムは正規言語 L を、live-complete な文字列集合 P と所属性神託を用いて学習します。ID アルゴリズムは P が live-complete であることを要請しますが、一方で ID アルゴリズムは P が大きくなると観察表が大きくなり必要な所属性神託の回数が増大するので、P を必要以上に大きくするのは良くありません。 本節では live-complete な文字列集合 P を実行時に生成する L* アルゴリズム [2] を説明します。L* アルゴリズムは次の二つの質問を神託に聞くことを通して学習を行ないます。現実的にこんなことがわかる神託をどうやって用意するのか、という問題はここでは考えません。

- 所属性質問 (membership query): 文字列 $w \in \Sigma^*$ を神託に与えて、w が学習したい言語 $L \subseteq \Sigma^*$ に含まれているか ($w \in L$) を問う質問。
- 等価性質問 (equivalence query): DFA $\mathscr{A}$ を神託に与えて、$L(\mathscr{A})$ が学習したい言語 L と等しいかどうかを問う質問。$L(\mathscr{A}) \neq L$ である場合には反例 $w \in \Sigma^*$、つまり $w \in L(\mathscr{A}) \triangle L$ を充たす最短の文字列 w が返る。ここで $\triangle$ は集合の対称差の記号である。

観察表が以下の条件を充たすとき、その観察表を**閉じている** (closed) と言います。

- 各 $w' \in P'$ について、$T[w,\text{-}] = T[w',\text{-}]$ を充たす $w \in P$ が存在する
- 各 $w' \in P$、$a \in \Sigma$ について、$w \cdot a \in P \cup P'$ が成り立つ

4.1. L* アルゴリズム

本節も例として、図 1 の NFA が認識する $L = \left\{a \in \{\text{K,B,F}\} \;\middle|\; \text{KBF を部分文字列として含む}\right\}$ を用います。

L* アルゴリズムではまず始めに図 6 左の観察表から始めます。最初に空欄になっているセル $T[\varepsilon,\varepsilon]$ を所属性質問を用いて埋め、同時に P' に K,B,F を追加し、同様に埋めます (図 6 右)。

ここで図 6 右の観察表が閉じているかを判定します。今回は任意の $w \in P'$ について、$T[w,\text{-}] = T[\varepsilon,\text{-}]$ が成り立ちますし、各 $a \in \Sigma$ について、$a \in P'$ が成り立つので観察表は閉じています。観察表が閉じている場合、ID アルゴリズムのときと同様に観察表から DFA を生成します。図 6 右の観察表からは図 7 の DFA、$\mathscr{A}_1$ が生成されます。

DFA $\mathscr{A}_1$ を生成することができたので等価性質問によって、DFA $\mathscr{A}_1$ が学習したい正規言語を認識するかどうかを調べてみます。今回、DFA $\mathscr{A}_1$ は言語 L を認識しないので、最短の反例 KBF が返ります。ここで、P に KBF の prefix で P に含まれていないものを

	ε
ε	-
K	-
B	-
F	-

図 6　一番始めの観察表 (左) と所属性神託を使って内容を埋めた観察表 (右)

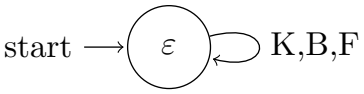


図 7　図 6 右の観察表に対応する DFA $\mathscr{A}_1$

	ε
ε	-
K	-
KB	-
KBF	+
B	-
F	-

図 8　K、KB、KBF を P に追加した観察表

追加します。KBF の prefix は ε、K、KB、KBF なので、今回は P に K と KB と KBF を追加し、観察表を埋めます。(図 8)

図 8 の観察表は閉じていないので、図 8 の観察表が閉じるまで P 及び P' に文字列を追加し、観察表を埋めます。(図 9 左) 具体的には $\mathrm{K}, \mathrm{KBKBF} \in P$ ですがこれらの一文字加えた文字のうち、KK、KF、KBK、KBB、KBFK、KBFB、KBFF が P にも P' にも含まれていないので、これらを P' に追加します。

次に図 9 左の観察表が一貫するまで S に文字列を追加します (図 9 右)。観察表を一貫させる操作は ID アルゴリズムのものと同じです。今回はここで観察表が閉じかつ一貫しましたが、一般には一度の操作で観察表が閉じかつ一貫するとは限りません。その場合、観察表が閉じかつ一貫するまで P' や S に繰り返し文字列を追加します。

最後に図 9 左の観察表に対応する DFA $\mathscr{A}_2$ を生成し、等価性質問によって $L\left(\mathscr{A}_2\right)$ と学習したい正規言語が一致するかどうかを調べます。今回は $L\left(\mathscr{A}_2\right)$ が学習したい正規言語と一致するので、L* アルゴリズムはここで終了となります。

	ε
ε	−
K	−
KB	−
KBF	+
B	−
F	−
KK	−
KF	−
KBK	−
KBB	−
KBFK	+
KBFB	+
KBFF	+

	ε	F	BF
ε	−	−	−
K	−	−	+
KB	−	+	−
KBF	+	+	+
B	−	−	−
F	−	−	−
KK	−	−	+
KF	−	−	−
KBK	−	−	+
KBB	−	−	−
KBFK	+	+	+
KBFB	+	+	+
KBFF	+	+	+

図 9　図 8 の観察表を閉じさせた観察表 (左)、及び更に一貫するまで S に文字列を追加した観察表 (右)

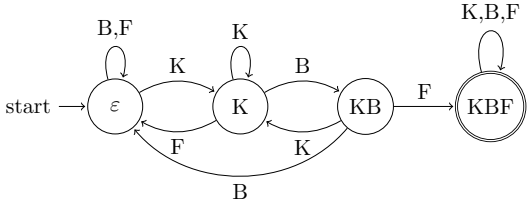


図 10　図 9 右の観察表に対応する DFA $\mathscr{A}_2$

4.2. 停止性

さて、Myhill-Nerode の定理の立場から L* アルゴリズムの観察表を見ていきます。 まず、 行ベクトル $T[w, \text{-}]$ が互いに異なるような $w, w' \in P$ については、 適宜末尾に文字列を加えたときの受理 / 非受理の関係が異なるので、$P/\approx_T$ の要素数は Σ^*/R_L の要素数以下になり、L が正規言語であるならば上限が存在します ($\approx_T$ の定義は 3 節を参照)。P に文字列を追加して観察表を閉じさせる操作を行うとき、任意の $w \in P$ について $T[w'a, \text{-}] \neq T[w, \text{-}]$ を充たすような $w'a$ (但し $w' \in P, a \in \Sigma$) を P に追加するので、$P/\approx_T$ の要素数は 1 増加します。観察表が一貫していないとき、S に要素を追加して、 つまり $\approx_T$ の定義を厳しくすることで任意の $w, w' \in P, a \in \Sigma$ について $T[w, \text{-}] = T[w', \text{-}]$ ならば $T[wa, \text{-}] = T[w'a, \text{-}]$ を充たすようにするので、$P/\approx_T$ の要素数は少なくとも 1 増加します。従って L が正規言語であるとき、観察表を閉じるまたは一貫させる操作は有限回しか行わ

れないことがわかります。また、等価性質問によって反例が返された後には少くとも一回以上 P に文字列を追加して観察表を閉じさせる、または観察表を一貫させる操作を行うので、等価性質問も有限回しか行われません。P' に文字列を追加して観察表を閉じさせる操作を行うのは等価性質問によって反例が返されたとき、または P に文字列を追加して観察表を一貫させる操作を行った後なので、これも有限回しか行われません。以上より、L が正規言語であるとき L* アルゴリズムが常に停止することがわかります。

また、L* アルゴリズムが生成する DFA の状態数は $\left|P/{\approx_T}\right|$ ですが、$L(\mathscr{A}) = L$ を充たす任意の DFA の状態数は Σ^*/R_L 以上であることから、$\mathscr{A}$ は $L(\mathscr{A}) = L$ を充たす状態数が最小の DFA であることもわかります。

5. NL* アルゴリズム

お母さん：「おばあちゃんが教えてくれたの！
L* が DFA しか学習できないのは悪魔のせいだって！

お母さん：「だからお母さん、 エクソシストさんを呼んであげたのよ！
必ず L* で NFA を学習してくれるから！

明美：「勝手に非決定的分岐を必要としてるんじゃねえ！
私は DFA で十分満足してるんだよ！

ガミジン：「その考え！人格が Myhill–Nerode に支配されている！

前節までに説明した ID アルゴリズムや L* アルゴリズムでは正規言語 L を表現する最小 DFA を学習します。一方で NFA を用いることで等価な正規言語を表現するオートマトンの状態数を指数的に小さくすることがあることが知られています。本節では L* アルゴリズムと同様に所属性質問と等価性質問を用いて、正規言語 L を表現する NFA を学習する、NL* アルゴリズムを紹介します。

5.1. Residual Finite-State Automata

ID アルゴリズムや L* アルゴリズムで観察表から DFA を生成できたカラクリは、観察表が一貫していて閉じているとき、観察表は (学習したい言語とは限らない) ある正規言語の Myhill-Nerode 同値関係を表現するので、これを用いて DFA を生成することができる、というものでした。同様に観察表を用いて NFA を学習しようとすると、何らかの方法で

NFA を生成する必要がありますが、このままの「一貫」や「閉」の定義のままでは最終的に Myhill-Nerode 同値関係が学習されてしまい、結局状態数を減らすことができません。NL* アルゴリズムでは ID アルゴリズムや L* アルゴリズムで用いた「一貫」や「閉」の代わりに RFSA 一貫 (RFSA-Consistency) と RFSA 閉 (RFSA-Closedness) を用い、観察表からは DFA の代わりに residual finite-state automaton (RFSA) と呼ばれる NFA のサブクラスを生成します。RFSA は DFA を含む NFA のサブクラスで、RFSA が表わす言語を DFA を用いて表わすと指数的に多くの状態数が必要となる場合があることが知られています。また、本章では省きますが RFSA には DFA での最小 DFA と同じ様に、canonical RFSA と呼ばれるものが存在します。L* アルゴリズムで最小 DFA が学習されるのと対応して、NL* アルゴリズムによって学習される RFSA は canonical RFSA になります。詳細は [4] を参照してください。

定義 6 (residual 言語)　言語 $L \subseteq \Sigma^*$ 及び $w \in \Sigma^*$ について、$w^{-1}L = \left\{w' \in \Sigma^* \;\middle|\; ww' \in L\right\}$ と定義する。言語 L' と言語 L について文字列 $w \in \Sigma^*$ で $L' = w^{-1}L$ を充たすものが存在するとき、L' が L の residual 言語であるという。言語 L の residual 言語の集合を $\mathrm{Res}(L)$ と表記する。 ◇

定義 7 (residual finite-state automata (RFSA))　NFA $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, \Delta\right)$ 及び $q \in Q$ について、$\mathscr{A}_q = \left(\Sigma, Q, \{q\}, Q_F, \Delta\right)$ 及び $L_q = L\left(\mathscr{A}_q\right)$ と表記する。NFA $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, \Delta\right)$ 及び任意の $q \in Q$ について、$L_q \in \mathrm{Res}\left(L(\mathscr{A})\right)$ を充たすとき、$\mathscr{A}$ が residual finite-state automaton であるという。 ◇

定義 8 ($\sqsubseteq$,$\sqcup$)　P, P', S を観察表 T の添字集合とする。観察表 T の行 $T[w,\text{-}]$ と $T[w',\text{-}]$ について、任意の $u \in S$ について $T[w,u] = + \Rightarrow T[w',u] = +$ が成り立つとき、$T[w,\text{-}] \sqsubseteq T[w',\text{-}]$ と表記する。観察表 T の行の添字 $w, w', w'' \in P \cup P'$ について、任意の $u \in S$ について

$$T[w,u] = + \Leftrightarrow T[w',u] = + \text{ または} T[w'',u] = +$$

が成り立つとき、$T[w,\text{-}] = T[w',\text{-}] \sqcup T[w'',\text{-}]$ と表記する。$w \in P \cup P'$ について、$w_1, w_2, \cdots, w_n \in \left(P \cup P'\right) \setminus \{w\}$ で

$$T[w,\text{-}] = \bigsqcup_{i \in \{1,2,\cdots,n\}} T[w_i,\text{-}]$$

を充たすものが存在しないとき、 行 $T[w, \text{-}]$ を**素**であるという。 観察表 T の素な添字集合を $\mathrm{Primes}(T) \subseteq P \cup P'$ と表記する。 ◇

定義 9 (RFSA 一貫)　P, P', S を観察表 T の添字集合とする。任意の $w, w' \in P$、$a \in \Sigma$ について、$T[w, \text{-}] \sqsubseteq T[w', \text{-}]$ ならば $T[wa, \text{-}] \sqsubseteq T[w'a, \text{-}]$ が成り立つとき、 観察表 T が **RFSA 一貫**であるという。 ◇

定義 10 (RFSA 閉)　P, P', S を観察表 T の添字集合とする。 任意の $w \in P'$ について、$T[w, \text{-}] = \sqcup\{T[w', \text{-}] \mid w' \in \mathrm{Primes}(T) \cap P, T[w', \text{-}] \sqsubseteq T[w, \text{-}]\}$ が成り立ち、任意の $w \in P$ 及び $a \in \Sigma$ について、$wa \in P \cup P'$ が成り立つとき、観察表 T が **RFSA 閉**であるという。 ◇

5.2. NL* アルゴリズム

本節では例として、図 11 の NFA が認識する言語 L を用いて NL* アルゴリズムの動作を見ていきます。

NL* アルゴリズムでも、L* アルゴリズムと同様にまず始めに図 12 左の観察表から始めます。 最初に空欄になっているセル $T[\varepsilon, \varepsilon]$ を所属性質問を用いて埋め、 同時に P' に K,B,F を追加し、 同様に埋めます (図 12 右)。 ここで図 12 右の観察表が RFSA 閉であるかを判定します。 今回は任意の $w \in P'$ について、$T[w, \text{-}] = T[\varepsilon, \text{-}]$ が成り立ちますし、各 $a \in \Sigma$ について、$a \in P'$ が成り立つので観察表は閉じています。

観察表が RFSA 閉である場合、観察表から RFSA を生成します。観察表から RFSA を生成する方法は以下の様に、ID アルゴリズムで DFA を生成した方法と似ています。

- 各 $w \in P \cap \mathrm{Primes}(T)$ に対して、RFSA の状態 q_w を生成する
- RFSA の初期状態は $Q_0 = \left\{q_w \;\middle|\; q_w \in Q, T[w, \text{-}] \sqsubseteq T[\varepsilon, \text{-}]\right\}$ とする。
- RFSA の受理状態は $Q_F = \left\{q_w \;\middle|\; q_w \in Q, T[w, \varepsilon] = +\right\}$ とする。

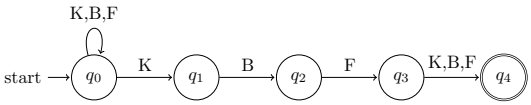


図 11　本章で学習する NFA。DFA に変換すると状態数が増加する。

	ε
ε	

	ε
ε	-
K	-
B	-
F	-

図 12　一番始めの観察表 (左) と所属性神託を使って内容を埋めた観察表 (右)

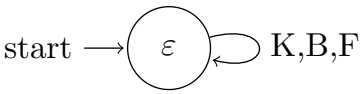


図 13　図 12 右の観察表に対応する RFSA $\mathscr{A}_1$

	ε	K	KB	KBF	KBFK
ε	-	-	-	-	+
K	-	-	-	+	+
B	-	-	-	-	+
F	-	-	-	-	+

図 14　K、KB、KBF、KBFK を S に追加した観察表

- RFSA の遷移関係 $\Delta \subseteq Q \times \Sigma \times Q$ は、$\Delta = \left\{ \left(q_w, a, q_{w'} \right) \;\middle|\; T[w', \text{-}] \sqsubseteq T[wa, \text{-}] \right\}$ とする。

図 12 右の観察表からは図 13 の RFSA、$\mathscr{A}_1$ が生成されます。

RFSA $\mathscr{A}_1$ を生成することができたので等価性質問によって、RFSA $\mathscr{A}_1$ が学習したい正規言語を認識するかどうかを調べてみます。今回、RFSA $\mathscr{A}_1$ は言語 L を認識しないので、最短の反例 KBFK が返ります。ここで、S に KBFK の suffix で S に含まれていないものを追加します。KBFK の suffix は ε、K、FK、BFK、KBFK なので、 今回は S に K、FK、BFK、KBFK を追加し、観察表を埋めます。(図 14)

	ε	K	FK	BFK	KBFK
ε	−	−	−	−	+
K	−	−	−	+	+
B	−	−	−	−	+
F	−	−	−	−	+
KK	−	−	−	+	+
KB	−	−	+	−	+
KF	−	−	−	−	+

図 15　K を P に、KK、KB、KF を P' に追加した観察表

	ε	K	FK	BFK	KBFK
ε	−	−	−	−	+
K	−	−	−	+	+
KB	−	−	+	−	+
B	−	−	−	−	+
F	−	−	−	−	+
KK	−	−	−	+	+
KF	−	−	−	−	+
KBK	−	−	+	+	+
KBB	−	−	−	−	+
KBF	−	+	−	−	+

図 16　KB を P に、KBK、KBB、KBF を P' に追加した観察表

	ε	K	FK	BFK	KBFK
ε	−	−	−	−	+
K	−	−	−	+	+
KB	−	−	+	−	+
KBF	−	+	−	−	+
B	−	−	−	−	+
F	−	−	−	−	+
KK	−	−	−	+	+
KF	−	−	−	−	+
KBK	−	−	+	+	+
KBB	−	−	−	−	+
KBFK	+	−	−	+	+
KBFB	+	−	−	−	+
KBFF	+	−	−	−	+

図 17　KBF を P に、KBFK、KBFB、KBFF を P' に追加した観察表

図 14 の観察表は閉じていないので、図 14 の観察表が閉じるまで P 及び P' に文字列を追加し、観察表を埋めます。(図 15 図 16 図 17 図 18)

最後に図 18 の観察表に対応する RFSA $\mathscr{A}_2$ を生成し、等価性質問によって $L(\mathscr{A}_2)$ と学習したい正規言語が一致するかどうかを調べます。今回は $L(\mathscr{A}_2)$ が学習したい正規言語と一致するので、NL* アルゴリズムはここで終了となります。

$\mathscr{A}_2$ は図 11 の NFA とは違った NFA ですが、認識する言語は等しいです。さらに、遷移関係の個数は多いですが、状態数は図 11 の NFA と同じで、同じ言語を認識する最小 DFA より小さいです。

5.3. L* アルゴリズムとの関係

L* アルゴリズムでの「閉」や「一貫」の定義は RFSA 閉や RFSA 一貫より強いので、L* アルゴリズムの意味で閉じていて一貫している観察表は NL* アルゴリズムの意味でも閉じていて一貫しています。また、NL* アルゴリズムでの観察表 T について、T が RFSA 閉かつ RFSA 一貫なとき、$w, w' \in \Sigma^*$ が wR_Lw' を充たすなら w と w' の少なくとも一つは素ではないので、T から生成された RFSA は同じ言語を認識する最小 DFA と状態数が同じか少なくなります。

本章では学習についての計算量解析や性能比較は基本的には省略するので、[4] を参照してください。NL* アルゴリズムでは所属性質問や等価性質問を行った直後に、素な添字の個数が減ることもあるので、停止性証明や計算量解析はやや込み入っています。また、計算量、つまり学習したい正規言語を学習するのに必要な所属性質問や等価性質問の回数を比較すると、最悪の場合は NL* アルゴリズムの方が L* アルゴリズムよりも多くなることが知られています。一方で実験的には L* アルゴリズムと比べて、NL* アルゴリズムはむしろ高速に動作することが報告されています。さらに、NL* アルゴリズムで学習される RFSA は、最小 DFA と比較してかなり状態数が小さくなることも、実験的に報告されています。

6. まとめ

川越市は今日も平和だった。

~ 完 ~

	ε	K	FK	BFK	KBFK
ε	−	−	−	−	+
K	−	−	−	+	+
KB	−	−	+	−	+
KBF	−	+	−	−	+
KBFB	+	−	−	−	+
B	−	−	−	−	+
F	−	−	−	−	+
KK	−	−	−	+	+
KF	−	−	−	−	+
KBK	−	−	+	+	+
KBB	−	−	−	−	+
KBFK	+	−	−	+	+
KBFF	+	−	−	−	+
KBFBK	−	−	−	+	+
KBFBB	−	−	−	−	+
KBFBF	−	−	−	−	+

図 18　KBFB を P に、KBFBK、KBFBB、KBFBF を P' に追加した観察表。この観察表は RSFA 閉である

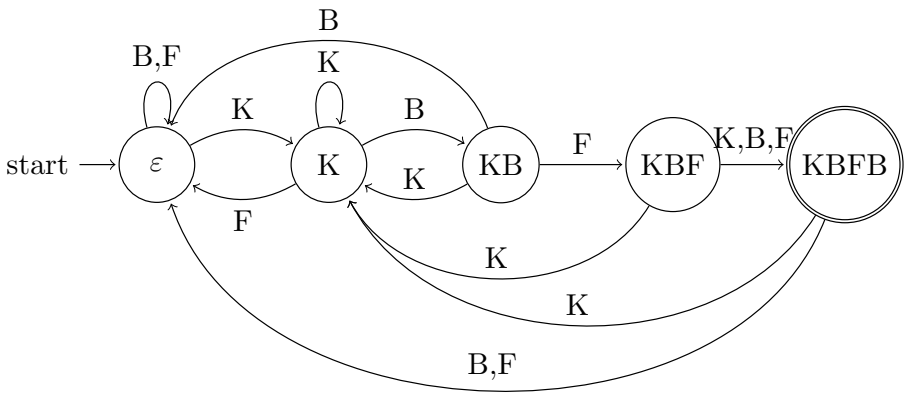


図 19　図 18 の観察表に対応する RFSA $\mathscr{A}_2$

参考文献

[1] J. ホップクロフト , R. モトワニ , J. ウルマン . オートマトン言語理論計算論 *I*. サイエンス社 , 2003.

[2] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2), pages 87–106, 1987.

[3] Dana Angluin. A Note on the Number of Queries Needed to Identify Regular Languages. *Information and Control*, 51(1), pages 76–87, 1981.

[4] Benedikt Bollig, Peter Habermehl, Carsten Kern, Martin Leucker. Angluin-Style Learning of NFA.. In *IJCAI*, pages 1004–1009, 2009.

[5] Michal Cadilhac. Review of handbook of weighted automata, edited by Manfred Droste, Werner Kuich and Heiko Vogler.. *SIGACT News*, 43(3), pages 32–37, 2012.

[6] Borja Balle, Mehryar Mohri. Learning Weighted Automata.. In *CAI*, pages 1–21, 2015.

[7] Loris D'Antoni, Margus Veanes. The Power of Symbolic Automata and Transducers.. In *CAV (1)*, pages 47–67, 2017.

[8] Samuel Drews, Loris D'Antoni. Learning Symbolic Automata.. In *TACAS (1)*, pages 173–189, 2017.

[9] Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen. A succinct canonical register automaton model.. *J. Log. Algebr. Meth. Program.*, 84(1), pages 54–66, 2015.

[10] Sofia Cassel, Falk Howar, Bengt Jonsson, Bernhard Steffen. Learning Extended Finite State Machines.. In *SEFM*, pages 250–264, 2014.

[11] Malte Isberner, Falk Howar, Bernhard Steffen. The Open-Source LearnLib - A Framework for Active Automata Learning.. In *CAV (1)*, pages 487–495, 2015.

[12] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, David R. Piegdon. libalf: The Automata Learning Framework.. In *CAV*, pages 360–364, 2010.

[13] Ali Khalili. *AIDE (Automata-Identification Engine) - CodePlex Archive*. `https://`

archive.codeplex.com/?p=aide, 2015.

[14] Karl Meinke. Learning-Based Testing: Recent Progress and Future Prospects.. In *Machine Learning for Dynamic Software Analysis*, pages 53–73, 2018.

Writing a (micro)kernel in Rust in 12 days - 2.5th day -

nullpo-head

1. 前回までのあらすじ

久しぶり！今回の Yabaitech.Tokyo でも、前号の続きとして Rust でのマイクロカーネル開発をやっていこう。この連載は、私が Rust でマイクロカーネルを開発する開発のログを、皆さんと共有するためのものだ。その過程で、Rust によるカーネル開発の最高な点や厄介な点を、皆さんにご紹介していきたい。そして、全ての開発が終わった暁には、たった 12 日間で Rust でカーネルを実装するための、ちょうどいいチュートリアルができあがっている（はず）だろう。[*9]前提知識としては、実際に自分でゼロからカーネルを書いてみたことはなくていいけれど、カーネルの仕組みのだいたいのイメージを把握していることを仮定している。CPU のプロテクションモード、と聞いて意味は分かるけれど、x86 の具体的なモードまでは覚えていない。アセンブリがどのようなものかなんとなくわかるけれど、あまり書いたことはない、とかその程度だ。本文中で、C 言語のコード、Makefile、それにリンカスクリプトだとかが簡単な説明だけで出てくることになる。詳細な説明こそはしないが、Wikipedia の一段落目に出てきそうな説明文くらいは書いてあるからある程度安心してほしい。

前号では、12 日間のうちの最初の一日目として、マイクロカーネルの歴史と L4 カーネルについて説明し、L4Ka::Pistachio カーネルをコンパイルして起動させてみたのだった。さらに、二日目では、Phillipp 氏の"Writing an OS in Rust"[3] に従って、Rust で小さな

9 これは希望的観測だ。そして、12 日間で終わるチュートリアルでも、チュートリアルそれ自体を書くことはとても 12 日間では終わらないようだ！

カーネル（と呼ぶことにしたベアメタルプログラム）を実装して、Qemu で動かした。このとき Philipp 氏の `bootloader` crate を用いてブートを実現したのだが、これが後々マイクロカーネルを作るにあたって不都合になってしまう。そこで、Multiboot 規格 [5] に沿ったブート方式にカーネルを書き直す必要があった。しかし、その部分で前回の連載は終了となってしまったのだった。なので今回は、この"2.5th day"を完成させていく記事になる。

2.5 日目に早速進む前に、少しマイクロカーネルの歴史と L4 カーネルについて振り返ってみよう。Mach に代表されるような伝統的なマイクロカーネルは、IPC のパフォーマンスに悪さに悩まされていた。マイクロカーネルでは、システムコールがサーバー間の複数の IPC で実装され、そして IPC はコンテキストスイッチを伴う。一方、モノリシックカーネルのシステムコールは通常 1 回のコンテキストスイッチで済むから、比較するとマイクロカーネルではシステムコールのコストが割高になってしまう。そこで、Jochen Liedtke 先生が最初に実装した L4 マイクロカーネルファミリーは、「極小性」をキーコンセプトとしてカーネルを再設計しなおし、パフォーマンスの高速化を図った。IPC の機構は Mach に比べて極端に単純になり、IPC のメッセージサイズは CPU のキャッシュに乗るよう再設計された。プロセスのスケジューリングも単純化され、IPC を高速化するいくつかの工夫も導入された。さらに初期のカーネルにいたってはすべてアセンブリで書いてしまうという執念の入りようで、ありうる極小のマイクロカーネルを完成させた。その結果として、L4 の IPC は Mach に比べて 20 倍も高速化したのだった。[1] [2]

そして、L4 の極小性は短期間でカーネルを自作してみるのにぴったりな性質だ！今回実装ターゲットにした L4 X.2 Standard では、定義されているシステムコールが 12 個しかない。この連載が、12 日でカーネルを作る、と言い張っているのは、API が 12 個しかない、というこの事実をあてにしているからだ。いま"Standard"と書いたように、L4 ファミリーには L4 カーネルが実装しているべきシステムコールやその ABI、割り込みハンドリングの規約などを定めた標準仕様が存在する。これはちょうど、普段慣れ親しんでいる POSIX 規約のカーネル版、とでもいえるものだ。これに準拠したカーネルを作れば、少ない修正で L4 カーネル用のユーザーランドを動かせるようになるというわけ。1 日目でビルドしてみた Pistachio カーネルは、この L4 X.2 Standard に準拠した L4 カーネルだった。具体的には、12 個のシステムコールのリストは次の通り。

(1) `KERNELINTERFACE`

(2) EXCHANGEREGISTERS

(3) THREADCONTROL

(4) SYSTEMCLOCK

(5) THREADSWITCH

(6) SCHEDULE

(7) IPC

(8) LIPC

(9) UNMAP

(10) SPACECONTROL

(11) PROCESSORCONTROL

(12) MEMORYCONTROL

冗談抜きでこれで全部だ。実にシンプルで小さい。このシステムコールさえ実装すれば、おおよそ Pistachio と互換のカーネルができあがることになる。そしてそれこそがこの連載の最終ゴールだ！

2. 2.5th Day: Minimal Kernel in Rust with Multiboot2

2.1. 前回まで

それでは 2.5 日目を始めよう。前号では Philipps' [*10]にしたがって、どこまでカーネルを実装していたのだろう？ GitHub 上のコードで言えば、このコミットのものまででであった。(https://github.com/nullpo-head/Rusty-L4/commit/72f400d7029a0431f87af27f64f-b40abb3ff68ee) これは、Philipp's の"VGA Text Mode" [*11]の章を終えたときの実装に相当する。これをビルドして、Qemu を使って出力されたバイナリファイルを起動すると、図 1 が表示される。このときは、VGA のメモリ領域に書き込んでメッセージを表示するような Free Standing な ELF ファイルを Rust で書いた。そして、その ELF のブートは Philipp 氏の bootloader crate と bootimage crate にお任せしたのだった。それにより、私

10 Philipp 氏の"Writing an OS in Rust"のことを、この連載では"Philipp's"と呼んでいる

11 https://os.phil-opp.com/vga-text-mode

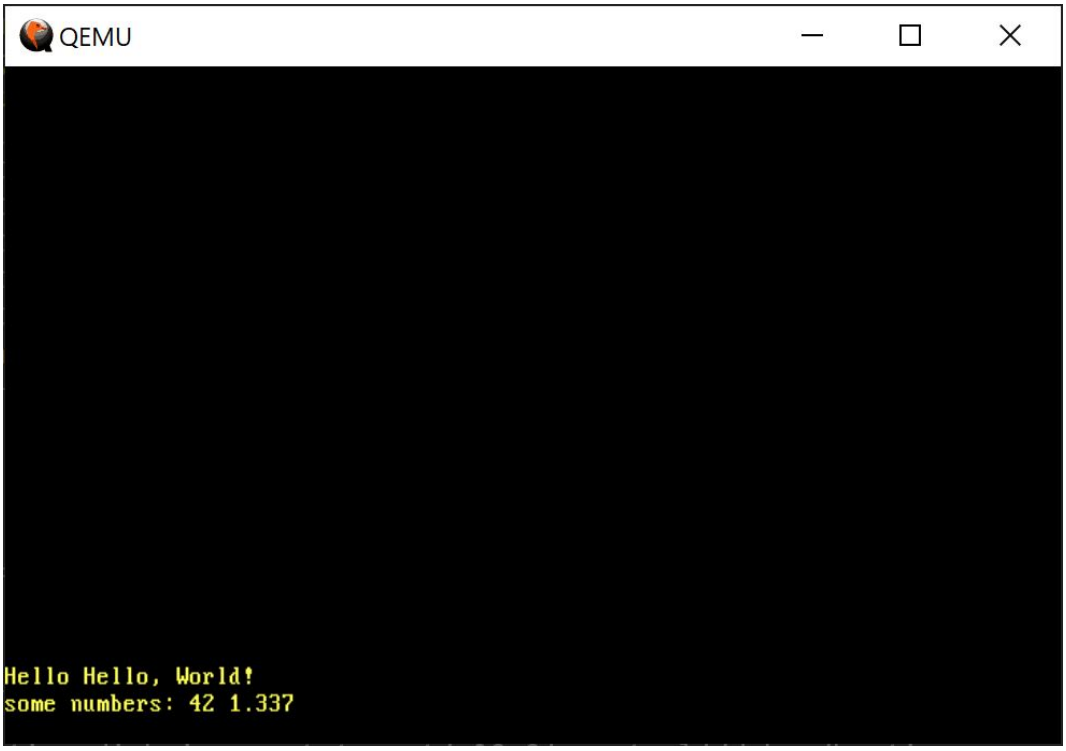


図 1　前号での最終画面

は自分自身でブートローダーを書くことなく、Rust プログラムを書くだけで x86_64 アーキテクチャのブートから、Rust ランタイムの初期化まで簡単に達成できたのだった。

2.2. Boot with Multiboot2

Bootloader Crate and Multiboot2

さて、では、その続きに進むとしよう！今回やることは、`bootloader` を使うことをやめて、Multiboot を使ったブートへと実装を変えることだ。なぜそんなことをする必要があるのだろう？その理由は 2 つある。

一つ目は、現状の `bootloader` crate が、マイクロカーネル構成と少し相性が悪いこと。マイクロカーネルは、ブートローダーに複数の ELF ファイルをマップしてもらえた方が実装がシンプルで済む。マイクロカーネルは、複数のユーザーランドで動くサーバー群があって初めてオペレーティングシステムとして動作するわけだけど、マイクロカーネ

ルのカーネル部分は非常にシンプルなため、ファイルシステムをパースして ELF ファイルをロードするような能力はない。だから、このままでは、マイクロカーネルはそもそもサーバー群をメモリに展開することができない。このような問題を解決するのが、まさに Multiboot の Boot Modules 機能だ。Multiboot の Boot Modules をサポートするブートローダーは、複数の ELF ファイルをブートイメージから取り出して、メモリ上に展開しておいてくれる。そして、カーネルを起動する際に、マップされた ELF の情報を所定の方法でカーネルに渡してくれる。逆に `bootloader` crate にはまだブートモジュールの機能はないため、少し工夫をしないとマイクロカーネルを起動することができないのだ。

二つ目の理由は、L4Ka::Pistachio の資産をスムーズに流用するためだ。Pistachio カーネルは、ちょうどいま説明した理由で Multiboot を使ってブートするようになっている。基本的に私は、1. まず Pistachio で動くサーバーを実装して Pistachio で動作を確認 2. その後そのサーバーが自分で実装したカーネルでも動作するようシステムコールを作りこむ、という順番で実装を進めていく予定だ。だから、Multiboot で自分のカーネルもブートするようにしておけば、Pistachio と自分のカーネルの差し替えがスムーズになるので、テストに都合がいい。この 2 つの理由から、Multiboot へと移行していこうと思う。

Write Multiboot2 Header, Linker Script, and Makefile

さて、もし最初の節で紹介した GitHub 上のコミットを clone してくれていれば、現在の Rusty L4 [*12]のディレクトリ構成は以下のようになるはずだ。前回の連載のコードをお手元にお持ちの方も、おおよそ似たような構成になっているだろう。

```
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── main.rs
│   └── vga_buffer.rs
└── x86_64-rusty_l4.json
```

今回は、このコードに変更を加えていく。

今回は、しばらく Rust の世界とお別れを告げることになる。快適でモダンだった Rust

12 前回、この連載で開発する Rust 製 L4 カーネルをそう名付けたのだった。

の世界から離れ、しばしアセンブリを地道に書いていくことになる。最終的なゴールは、アセンブリの世界から Rust の世界へと舞い戻ることだ。まず、初めに `startup.S` を加えよう。今までは `bootloader` crate と `bootimage` crate に頼っていた、ブートから Rust の関数にジャンプするまでの道のりを、このアセンブリファイルで置き換えることになる。このファイルの先頭に早速書かなければならないのが、Multiboot ヘッダーだ。

```
  .section .multiboot_header
  .balign  16
_mb_header:
  .long 0xe85250d6   /* magic */
  .long 0x00000000   /* flags */
  .long _mb_header_end - _mb_header /* header length */
  .long 0x100000000 - (0xe85250d6 + 0 + (_mb_header_end - _mb_header))
  .word 0x0
  .word 0x0
  .long 0x8
_mb_header_end:
```

Multiboot の仕様に従ったカーネルを書くのは非常に簡単で、 今書いたこの構造体が、カーネルイメージファイルの先頭 32768 バイトのどこかに、64bit アラインされた状態で埋め込まれているだけでいい。[5] もしカーネルイメージが ELF フォーマットでない場合はもう少しメモリレイアウトに関する情報を書かなければならないが、ELF フォーマットを使うなら、メモリレイアウトも ELF をパースして勝手に読み取ってくれる。 上の構造体のフィールドに関する詳しい説明は、興味があれば仕様を参照してくれればわかるけど、基本は上のものをコピーして使いまわせば OK だ。 逆に気を付けなければいけないことは、Multiboot のバージョンだ。Multiboot にはバージョン 1 とバージョン 2 があって、今まさに書いたのはバージョン 2 の方。実は前号では Multiboot1 を使ったのだけれど、 今回は 2 へと変更している。前号でビルドした Pistachio カーネルの方も、のちほど Multiboot2 を使うように書き直そう。どうしてバージョン 2 が必要かというと、ELF64 ファイルのブートを簡単にできるからだ。 バージョン 1 で 64bit ELF をブートするのは困難が伴ってしまう。そういう訳で、今回使うのはバージョン 2 の方になる。

次に、 ブートされて最初に実行されるアセンブリを書いていこう。Multiboot 対応のブートローダーによってブートされたカーネルは、x86CPU では 32bit Protected モードでブートローダーから処理が回ってくる。このモードでは、CPU は 32bit モードで動いてい

て、セグメントベースのメモリ保護機能は有効になっている。しかし、ページング、つまり仮想アドレスは有効にされていない。 普段使っている 64bit OS では、CPU は 64bit モードで動いていて、 仮想アドレスも有効だ。 このモードは 64bit ロングモードという。 今から書くスタートアップスクリプトは、32bit Protected モードから処理をはじめ、 最終的に 64bit ロングモードに入り、Rust のエントリー関数を呼び出すことが目的だ。 それをやるまでのステップは次のようになる。

(1) 仮のスタックを用意する。

(2) Rust で本格的にメモリ管理を行うまでの間に使う、仮のメモリレイアウトでページングを有効にする。

(3) Compatibility Long モードを有効化する。

(4) 仮の GDT を用意する。

(5) 64bit Long モードを有効化する。

(6) Rust のエントリー関数へジャンプする。

これを順番にやっていくことになる。 まずは仮のスタックを用意して、 シリアルポートからデバッグ出力を行う関数を作って動作を確認することから始めよう。

```
###
# The multiboot entry point
###
  .text
  .globl _start
.code32
_start:
  cli  // 割り込みの無効化
  cld  // DF をクリア
  movl $_mini_stack, %esp
# Debug
  movb $'A', %al
  call _putc_serial
  hlt

_putc_serial:
  movl $0x3f8, %edx
  outb %al, %dx
```

```
  ret

###
# Stack
###
  .data
  .space 1 << 21
_mini_stack:
```

上のコードは、エントリーポイント、デバッグ用のシリアル出力関数、そして Rust でメモリ管理周りを書くまでお世話になるスタック領域から成っている。ELF では慣習的にエントリーポイントを `_start` シンボルにするのでそれに従う。だから、私のアセンブリスクリプトでも、最初に実行される場所はここだ。まず、`_mini_stack` に確保したスタック領域に `esp` を初期化する。ところでこのスタックは、2MB という初期スタックとしては結構大きいサイズを使っているんだけれど、これは Rust のコードを正しく動かすのに非常に重要なことなのだ。Rust では、ベアメタルでもかなりたくさんの標準ライブラリや crate が使えることを前回の実装で見たと思う。結果として、C と違って関数を呼び出す際に消費されるスタックがかなり深くなるうえに、ライブラリでどの程度スタックが消費されるのかの予想が難しい。例えば、Deref なんかが走る際にもきっちりスタックが消費されていくのだ。だから、あらかじめかなり大きいサイズのスタックを使っておくと、後々奇妙なバグに悩まされることがなくなるだろう。そのため、ここは学習のためにカーネルを書いている、と割り切って富豪的に割り当てておくことにした。

`_putc_serial` はデバッグのためのシリアルポート出力のための関数だ。`call` 命令を使って呼び出しているため、用意したスタックがきちんと消費できるかのスタックのテストも兼ねている。`call` 命令は戻りアドレスをスタックの先頭に積むからね。`_putc_serial` では、シリアルポートを初期化することなく、いきなり 0x3f8 + 5 番に出力している。これは本来はおかしなコードだが、Qemu では、デバッグのために、初期化せずにシリアルポートをたたいても出力がうまくいくようになっている。このコードはそれに甘えさせてもらっている。

せっかく書いたアセンブリだが、まだビルドするにはパーツが足りない。`startup.S` を追加したら、次はリンカスクリプトだ。このあと、さらにビルドスクリプトを兼ねた

Makefile を追加して、極小カーネル、とまではいえないけど、ベアメタルプログラムである、`startup.S` をビルドしよう。では、`linker.ld` を加えよう。

```
ENTRY(_start)

SECTIONS {
    . = 1M;

    .text :
    {
        *(.multiboot_header)
        *(.text*)
    }

    .rodata :
    {
        *(.rodata*)
    }

    .data :
    {
        *(.data*)
    }

    .bss :
    {
        *(.bss*)
    }
}
```

リンカスクリプトは、本当にベアメタルプログラミングをやったことがある人にしか馴染みがないだろう。リンカがオブジェクトファイルのリンクを行う際、普段は何も指定しなくても平凡な ELF ファイルやそのほかの実行バイナリを出力してくれる。この実行バイナリの作成を細かく制御するのに使えるのがこのリンカスクリプトだ。リンカスクリプトに関する資料は本当に少なく、ほとんど GNU Ld のマニュアルである、"Using ld, the GNU Linker"くらいしかない。見た目がきれいなため、ここでは Redhat にホストされているバージョンを紹介しよう [6]。しかし、これだけを読んでも特段リンカスクリプトが書けるようになるわけではない。リンカスクリプトの使い方を理解するための、私が把握

している唯一の方法は、既存のプロダクトのリンカスクリプトを読むことだ。xv6 のような小さな OS のリンカスクリプトは、とっかかりとして最高だ。意外と色々と網羅的に書かなくても、なんとなくリンクが成功することが分かるだろう。LLVM プロジェクトのリンカである、lld の作者、rui さんに聞いたところによると、リンカはリンカスクリプトがセクションを網羅していなくても、良い感じに推測して配置してしまうらしい。ちなみに、ここらへんの仕様はあいまいで、undocumented なんだってさ。[*13] xv6 のリンカスクリプトとは逆に、GCC で暗黙的に使われているデフォルトのリンカスクリプトはかなり網羅的に書かれている。[7] 一度見てみると勉強になると思う。

さて、いま書いたリンカスクリプトは、かなりシンプルなものだ。実際にリンカスクリプトを読者のあなたが書く際には、どちらにせよ先ほど紹介したドキュメントを一読しないと右も左も分からないだろうから、ここでは雰囲気をつかんでもらうための説明に留めよう。

話を簡単にするために ELF フォーマットのことだけを考えよう。a.out や PE ファイルもおおよそ似たようなものだ。そもそも、ELF では、プログラムのコードはいくつかのセクションというものに分かれている。そして代表的なセクションが、リンカスクリプト上にもある `.text`, `.rodata`, `.data`, そして `.bss` だ。それぞれ、普通の実行コード、定数値、初期値付きグローバル変数、そしてグローバル変数だが初期値が 0 のもの、が代表的な中身になる。もちろんこれらの分類は、完全に各プログラミング言語のコンパイラ依存だから、あくまで、代表的な中身の話だ。また、セクションの名前自体も代表的な名前なだけで、実際は可変だし、さらにほかにも代表的なセクションはたくさんある。

そして、リンカに与えられるオブジェクトファイル群の、そういったセクションを、どのようにまとめたうえで、どこのメモリアドレスに配置するか制御できるのがリンカスクリプトだ。今回書いたリンカスクリプトは、まず、エントリーポイントを標準的 `_start` シンボルに設定している。その後入力ファイルのセクションの配置について記述していく。最初の部分で、まず、各セクションの配置の開始アドレスを、少なくとも 1M 以降になるよう現在アドレスを更新する。これは x86 アーキテクチャでのメモリホールを避けるためだ。そのあとはシンプルに、代表的なセクションを配置している。漏れたセクションは良い感じに配置されるので実はこの部分の大半は書かなくても大丈夫だったりする。

13 https://twitter.com/rui314/status/1107777197689835520

大事なのは、最終出力に含まれる .text セクションに、.multiboot_header を含めていることだ。これにより、マルチブートヘッダが text セクションの最初に配置されることになる。結果として、マルチブートヘッダが ELF ファイル上でも先頭の方に配置されることになり、Multiboot 対応ブートローダが、この ELF バイナリをカーネルとしてロードできるようになる。

詳しい人はここで少し違和感を感じるかもしれない！どうしてマルチブートヘッダを text セクションの中にいれるの？新しくセクションを作ったほうがいいんじゃない？と。実は後日リンカを lld に変えるさい、それだとマルチブートヘッダがファイルの後方に配置されてしまい、うまくいかなくなる。GCC はメモリ上に実際にロードされないセクションを ELF ファイル上で先頭の方に配置するけれど、逆に lld は後方に配置するのだ。結果として、ブートローダがマルチブートヘッダを見つけられなくなってしまう。その対策として、ここでは text セクションにマルチブートヘッダを入れてしまったのだが、後日マルチブートヘッダ単体でコンパイルして、先頭にくっつける方針に変えようかと思う。だけどひとまず、今日のところはこの方針で行く。

さぁ、ビルドに必要な最後のパーツ、ビルドスクリプトを兼ねた Makefile を作ろう。

```
all: run

build/kernel: src/linker.ld src/startup.S | build/
  gcc -fno-pic -no-pie -nostdlib -Tsrc/linker.ld -o build/kernel -Wl,-n sr
c/startup.S  # - (1)

build/:
  mkdir build

build/os.iso: build/kernel grub.cfg  # - (2)
  mkdir -p build/isofiles/boot/grub
  cp build/kernel build/isofiles/boot/kernel.bin
  cp grub.cfg build/isofiles/boot/grub/
  grub-mkrescue -o build/os.iso build/isofiles

run: build/os.iso
  qemu-system-x86_64 -cdrom build/os.iso -serial stdio -cpu Haswell,+pdpe1g
b -no-reboot  # - (3)
```

```
clean:
  rm -rf build
  cargo clean

.PHONY: run all clean
```

(1) で、先ほどまでに用意した startup.S と linker.ld のアセンブルとリンクを行っている。gcc に先ほど書いたリンカスクリプトとアセンブリを渡して、最終的に `build/kernel` へと出力している。 ここで `-fno-pic -no-pie` を渡しているのは、PIE バイナリの出力を抑制するためだ。最近の gcc や clang では、セキュリティの向上のためデフォルトで Position Independent Executable を出力するのだけど、この機能は配置アドレスを指定してリンクしたい私達の要求と衝突する。 リンクが通らなくなるので必ず無効化しよう。KASLR のような高度な機能は、今のところはスコープの範囲外だ。

これで、ブートすべきカーネルに当たる部分のビルドはできるようになった。では、次はこれをブートするブートローダを用意する番だ！今回は、Multiboot2 をサポートするメジャーなブートローダである grub2 を用いることにした。最近の PC だと、grub がインストールされていないことも割とあるだろうから、grub の周辺ツールを使うため、まずはインストールから始めよう。私の環境にいたっては WSL だから、もちろん grub は入っていない。以下のコマンドで必要となるツールをインストールしよう。

```
# apt install grub-common grub-pc-bin xorriso
```

これでインストールできるツールを使って、grub をつかったブータブルイメージを作成する。

(2) の部分を参照してほしい。この Make ルールで、`grub-mkrescue` コマンドを使って、grub を使ってブータブルになった、iso イメージを作成する。`grub-mkrescue` は、 次のようなディレクトリを入力すると、それをブータブルな iso イメージにしてくれる。

```
.
└── boot
    ├── grub
    │   └── grub.cfg
```

```
    └── kernel.bin
```

このとき、grub-mkrescue は、ディレクトリ中の grub.cfg からブートの構成を読み取ってイメージを作成する。そのための grub.cfg を、プロジェクトのルートディレクトリに追加してほしい。Makefile がこれを build/isofiles にコピーするようになっている。

```
set timeout=0
set default=0

menuentry "rusty l4" {
    multiboot2 /boot/kernel.bin
    boot
}
```

これで、完了だ！あとは make をするだけだ！その前に一度、確認しやすいよう、現在のディレクトリ内容を載せておこう。

```
.
├── Cargo.lock
├── Cargo.toml
├── Makefile
├── grub.cfg
├── src
│   ├── linker.ld
│   ├── main.rs
│   ├── startup.S
│   └── vga_buffer.rs
└── x86_64-rusty_l4.json

1 directory, 9 files
```

確認が済んだら make しよう。ルールにより、(3) の run が実行され、Qemu 上でブータブルディスクが実行される。シリアル出力が標準出力にリダイレクトされるので、start-up.S で書いたデバッグ出力、'A'が無事出力されれば成功だ。実際に実行してみると次のようになる。

```
$ make
mkdir build
gcc -fno-pic -no-pie -nostdlib -Tsrc/linker.ld -o build/kernel -Wl,-n src/
startup.S
mkdir -p build/isofiles/boot/grub
cp build/kernel build/isofiles/boot/kernel.bin
cp grub.cfg build/isofiles/boot/grub/
grub-mkrescue -o build/os.iso build/isofiles
xorriso 1.4.8 : RockRidge filesystem manipulator, libburnia project.

Drive current: -outdev 'stdio:build/os.iso'
Media current: stdio file, overwriteable
Media status : is blank
Media summary: 0 sessions, 0 data blocks, 0 data, 69.8g free
Added to ISO image: directory '/'='/tmp/grub.niGQlv'
xorriso : UPDATE : 284 files added in 1 seconds
Added to ISO image: directory '/'='/mnt/c/Users/abctk/Downloads/rusty_l4/2R
usty-L4/build/isofiles'
xorriso : UPDATE : 288 files added in 1 seconds
xorriso : NOTE : Copying to System Area: 512 bytes from file '/usr/lib/grub/
i386-pc/boot_hybrid.img'
xorriso : UPDATE :  54.62\% done
ISO image produced: 3475 sectors
Written to medium : 3475 sectors at LBA 0
Writing to 'stdio:build/os.iso' completed successfully.

qemu-system-x86_64 -cdrom build/os.iso -serial stdio -cpu Haswell,+pdpe1gb -
no-reboot
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01
H:ECX.fma [bit 12]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01
H:ECX.pcid [bit 17]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.
01H:ECX.x2apic [bit 21]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.
01H:ECX.tsc-deadline [bit 24]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.
01H:ECX.avx [bit 28]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01
H:ECX.f16c [bit 29]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01
```

```
H:ECX.rdrand [bit 30]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.
07H:EBX.hle [bit 4]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.0
7H:EBX.avx2 [bit 5]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.07
H:EBX.invpcid [bit 10]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.0
7H:EBX.rtm [bit 11]
A
```

かなり長い出力の最後に印字されている、小さな'A'に注目してほしい！この短い出力が、私の書いた小さなカーネルがきちんとブートされている証拠だ。一度にたくさん書くことになってしまったが、無事 `startup.S` がベアメタルで動いているようだ！

Jump to Rust

これで無事にブートができたわけだけど、 実はやることはまだある。 最初の方で、`startup.S` がやらなければならないことのステップをこう書いた。

(1) 仮のスタックを用意する

(2) Rust で本格的にメモリ管理を行うまでの間に使う、仮のメモリレイアウトでページングを有効にする。

(3) Compatibility Long モードを有効化する。

(4) 仮の GDT を用意する。

(5) 64bit Long モードを有効化する。

(6) Rust のエントリー関数へジャンプする。

そして、振り返ってみると終わったのはまだ、(1) だけだ。ロングモードの世界に入って、さっさと Rust の世界に帰り、今日の作業を終わることにしようじゃないか。

まず、 ページングの有効化だ。個人的には、 アセンブリのまま書くこの部分がいつも一番面倒だ！ x86 アーキテクチャのページング、 つまり仮想アドレス機構については語ることが多いが、 アセンブリのままではあまり説明が分かりやすいようなコードにはならない。 そのうえ、 当分のコードに必要なページ分だけ、 仮想アドレスと物理アドレス

が等しくなるようなマッピングをここではするだけだから、その説明もまだ生きてこない。そういうわけで、一旦ここは初期化のためのコードをひとまず受け入れてもらうことにしよう。x86 のページングに関する詳しい説明は、Rust の世界で本格的なページマッピングを作る際に回すことにする。

```
###
# The multiboot entry point
###
  (前回と同じなので中略)
  movb $'A', %al
  call _putc_serial
# Setting up the initial paging
  /* Set PAE */
  movl %cr4, %eax
  orl $1 << 5, %eax            /* PAE */
  movl %eax, %cr4
  movl $0x80000001, %eax       /* implicit argument for cpuid */
  cpuid
  andl $1 << 26, %edx          /* Test if it supports giga byte page */
  jnz 1f
  movl $_nohugepage_msg, %eax
  call _puts_serial
  hlt
  _nohugepage_msg:
  .ascii "\nNo, huge page support. halting..\n\0"
1:
  /* Make pml4 point to pdp */
  movl $boot_pml4, %ebx
  movl $boot_pdp, %eax
  orl $0x3, %eax              /* Present and RW */
  movl %eax, 0(%ebx)
  xorl %eax, %eax
  movl %eax, 4(%ebx)
  /* Put an indentity-mapping pdp entry */
  movl $boot_pdp, %ebx
  movl $0x83, %eax            /* Present, RW, and Page Size */
  /*   - the 0th entry to map [0, 1G) to [0, 1G) */
  movl %eax, 0(%ebx)
  xorl %eax, %eax
  movl %eax, 4(%ebx)
```

```
  /* Set them to cr3 */
  movl $boot_pml4, %eax
  movl %eax, %cr3
# Enter Long mode
  movl $0xc0000080, %ecx  /* EFER MSR */
  rdmsr
  orl $1 << 8, %eax       /* Set LME */
  wrmsr
  movl %cr0, %eax
  orl $1 << 31, %eax      /* Enable paging */
  movl %eax, %cr0
# Debug
  movb $'B', %al
  call _putc_serial
```

```
_puts_serial:
  movl $0x3f8, %edx
  movl %eax, %ebx
1:
  movb (%ebx), %al
  testb %al, %al
  jz 2f
  outb %al, %dx
  incl %ebx
  jmp 1b
2:
  ret

###
# Paging Structures
###
  .balign  4096
  .globl boot_pml4
boot_pml4:
  .space 4096, 0
  .globl boot_pdp
boot_pdp:
  .space 4096, 0
  .globl boot_pd
```

```
boot_pd:
  .space 4096, 0
```

注意点だけいくつか説明したい。まず、ページの初期化を簡単にするため、1GB ページを使って、ざっくり仮想アドレスの [0, 1G) を、物理アドレスの同じく [0, 1G) の空間へとマッピングした。実機では、1GB ページングの機能は Haswell 以降の比較的新しい CPU にしか載っていないため注意してほしい。また、このときにつかう、ブート時の仮ページング構造体群のアドレスを、.space ディレクティブを使って確保している。これは ELF ファイル上でも 4096 バイトの空間を消費してしまうので、本来なら .comm の方が好ましいのだが、ここを .comm にしているとなぜかうまくページングが有効化できなくなる。私はまだこの原因が良く分かっていないので要調査だ。だから、ここでは私を信用せずに、半信半疑の状態で .space を使用してほしい。

さて、ここまでのコードで、私達は Compatibility Long モードへと突入した。x86_64 アーキテクチャで導入されたロングモードだけど、実は 2 つのサブモードがある。そのうち一つは、本当の 64bit モードだけど、もうひとつはこの Compatibility Long モードなのだ。この状態では、まだ CPU は 32bit で動いている。ここで、64bit ロングモード用の Global Descriptor Table (GDT) をセットして、現在の 32bit コードから 64bit コードの領域へと long jump を行えば、本当の 64bit モードで CPU が動き出す。そうすれば、ようやく Rust へと処理を渡すことができる。

では、GDT を作っていこう。GDT は、64bit では、ほとんどその機能が廃止されたセグメントディスクリプタについて設定するための構造体だ。32bit 時代では、セグメントを使って、論理アドレスのオフセットやサイズを制限することができたのだけれど、今ではこの機能はなくなってしまった。権限など、必要な数少ないビットだけをセットしたら、あとはあまり用はない。詳しくは Intel のマニュアルを参照してほしい。ここでは、OS Dev Wiki に従って GDT を初期化する。

```
###
# GDT
###
  .data
  .align 4096
_boot_gdt:
```

```
  .set _boot_gdt_null, . - _boot_gdt
    /* null */
  .short 0xffff               /* Limit (low) */
  .short 0x0000               /* Base (low) */
  .byte  0x00                 /* Base (middle) */
  .byte  0x00                 /* Access */
  .byte  0x01                 /* Granularity */
  .byte  0x00                 /* Base (high) */
    /* code */
  .set _boot_gdt_code, . - _boot_gdt
  .short 0x0000               /* Limit (low) */
  .short 0x0000               /* Base (low) */
  .byte  0x00                 /* Base (middle) */
  .byte  0b10011010           /* Access; E/R */
  .byte  0b10101111           /* Granularity; 64 bits flag, limit19:16 */
  .byte  0x00                 /* Base (high) */
    /* data */
  .set _boot_gdt_data, . - _boot_gdt
  .short 0x0000               /* Limit (low) */
  .short 0x0000               /* Base (low) */
  .byte  0x00                 /* Base (middle) */
  .byte  0b10010010           /* Access; R/W */
  .byte  0x00                 /* Granularity */
  .byte  0x00                 /* Base (high) */
  .quad 0
_boot_gdtr:
  .word . - _boot_gdt - 1
  .quad _boot_gdt
```

そうしたらあとは、この GDT をロードして、64bit で書いたコードの領域へジャンプしよう。この 64bit コード領域が、さらに Rust のコードへとジャンプすることになる。

```
  movb $'B', %al
  call _putc_serial
# Load GDT
  movl $_boot_gdtr, %eax
  lgdt (%eax)
    # Debug
  movb $'C', %al
```

```
  call _putc_serial
  ljmp $_boot_gdt_code, $jump_to_rust
  hlt

###
# Long mode entry point
###
  .section .text
  .globl jmp_to_rust
.code64
jump_to_rust:
  cli
  cld
# Debug
  movb $'D', %al
  call _putc_serial
  hlt
```

これでロングモードへ突入だ！一旦ここで動作を確認してみよう。

```
$ make
gcc -fno-pic -no-pie -nostdlib -Tsrc/linker.ld -o build/kernel -Wl,-n src/
startup.S
mkdir -p build/isofiles/boot/grub
(中略)
ABCD
```

無事、デバッグ出力が'A'から'D'まで出力されている！ロングモードへの遷移の成功だ。残りのやることは、Rust のエントリーポイントへのジャンプだけだ。

だがその前に、久しぶりに Rust 側の世界にちょっとだけ戻ってこよう。Rust をアセンブリから呼べるようにするために、ちょっとだけやることがある。まず、`bootloader` crate を依存関係から削除する。いまや、私は Multiboot を使ってブートを行っているから、この crate はもう必要ない。 さらに、crate type を `staticlib` に書き換える必要がある。 これにより、cargo xbuild による出力が `.a` ファイルになり、自分でリンクを行うのに都合がよくなる。`Cargo.toml` を次のように書き換える。

```
+[lib]
```

```
+crate-type = ["staticlib"]
+
[dependencies]
-bootloader = "0.3.12"
```

さらに、main.rs も次のように修正する。

```
 #![no_std]
-#![no_main]
(中略)
 #[no_mangle]
-pub extern "C" fn _start() -> ! {
+pub extern "C" fn rust_start() -> ! {
```

この main.rs は、crate type の変更の影響で、lib.rs にリネームする必要がある。

```
$ git mv src/main.rs src/lib.rs
```

これで、Rust 側のコードを、今まで書いていたコードとリンクし、アセンブリから呼び出す準備が整った。

startup.S を完成させよう！

```
# Debug
  movb $'D', %al
  call _putc_serial
# Set data segment
  movw $_boot_gdt_data, %ax
  movw %ax, %ds
  movw %ax, %es
  movw %ax, %fs
  movw %ax, %gs
  movw %ax, %ss
# Fill the screen with blue
  movl $0xB8000, %edi
  movq $0x1F201F201F201F20, %rax
  movl $500, %ecx
  rep stosq
```

```
# Debug
  movb $'E', %al
  call _putc_serial
  movb $'\n', %al
  call _putc_serial
  jmp rust_start
  hlt
```

ロングモードに入ったあと、ds などのデータ用セグメントレジスタを先ほど初期化した GDT に設定していく。cs レジスタに関しては、ここに飛んでくる long jump で設定されている。そうしたらあとは rust_start へとジャンプするだけなのだが、前回の画面と変わりがないのも悲しいので、画面を青く塗りつぶしてみた。これで startup.S は完成だ！

これですべておしまい、と言いたいところなんだけど、かなり大事なステップが残っている。Rust のオブジェクトをリンクして、ブータブルイメージを作らなければいけない。そこで、Makefile を次のように変更しよう。

```
build/kernel: build/startup | build/
  cargo xbuild --target x86_64-rusty_l4.json  # - (1)
  cp target/x86_64-rusty_l4/debug/librusty_l4.a build/
  ld -n -Tsrc/linker.ld -o build/kernel-nonstripped build/startup build/libr
usty_l4.a # - (2)
  objcopy -g build/kernel-nonstripped build/kernel # - (3)

build/startup: src/linker.ld src/startup.S | build/
  gcc -c -fno-pic -no-pie -nostdlib -Tlinker.ld -o build/startup -Wl,-n sr
c/startup.S

(中略)
.PHONY: run all clean build/kernel
```

さきほどと比べると、startup.S のビルドが分離し、(1) に cargo xbuild によるビルドが追加されている。気を付けてほしいのは (2)、(3) の部分だ。(2) でリンクをしたあと、(3) で少し奇妙なことをしている。リンクが終わった後、kernel-nonstripped では Rust 由来のメタデータが ELF ファイル上でマルチブートヘッダより前方に位置してしまい、grub がマルチブートヘッダを見つけられなくなってしまう。そこで (3) のようにメタデータを strip することで、この問題を解決している。これは、根本的にはマルチブートヘッダの

ELF ファイル上での配置を実装依存の挙動に任せていることが問題だ。 前に言ったように、 これは後日なんらかの方法で改善したいと思っている。Garasubo 氏が開発している、erkos という素晴らしい Rust 製の組み込み OS がある。[9] これのコードベースを参考にさせてらったところ、cargo に直接リンカスクリプトを渡すこともできるようだ。 これを利用して cargo の出力するバイナリのセクション配置を制御することもできるだろうし、やはりマルチブートヘッダバイナリを単にバイナリ先頭にリンクすることもできるだろう。これは後日の課題にしよう。

なにはともあれ、これですべての準備が整った。make して実行しよう！前号からコードを手元で書いてくださっている方ではない、今回から git clone して試してみた、という方は、 おそらく `cargo xbuild` のインストールが終わっていないだろう。 インストールしたのち、さらに Rust L4 のビルドが nightly で行われるよう cargo を設定してほしい。

```
$ cargo install cargo-xbuild
$ rustup override add nightly
```

では、make しよう！

```
$ make
cargo xbuild --target x86_64-rusty_l4.json
(中略)
ABCDE
```

出力はこのようになるだろう。 そして、 図 2 のような画面が描画される！ Philipp の高級な crate を使ってブートした時と同じ Hello, World(をちょっとカラフルにしたもの) が無事表示できた。これで、2nd Day の Rusty L4 の最小カーネルは今度こそ完成だ！

2nd Day: Summary

二日目の Rust で最小のカーネルを書いてみる章は、 前号と今号にまたがった記事となってしまった。前号の前半では、Rust でのベアメタルプログラミングの世界を体験することが、大きな比重を占めた。最終的には、Qemu の画面上に、"Hello, World!"を表示するわけなんだけど、この際、ベアメタルプログラミングをしているにもかかわらず、crate を使って簡単にブートができてしまったり、println! マクロが使えるまであっという間だっ

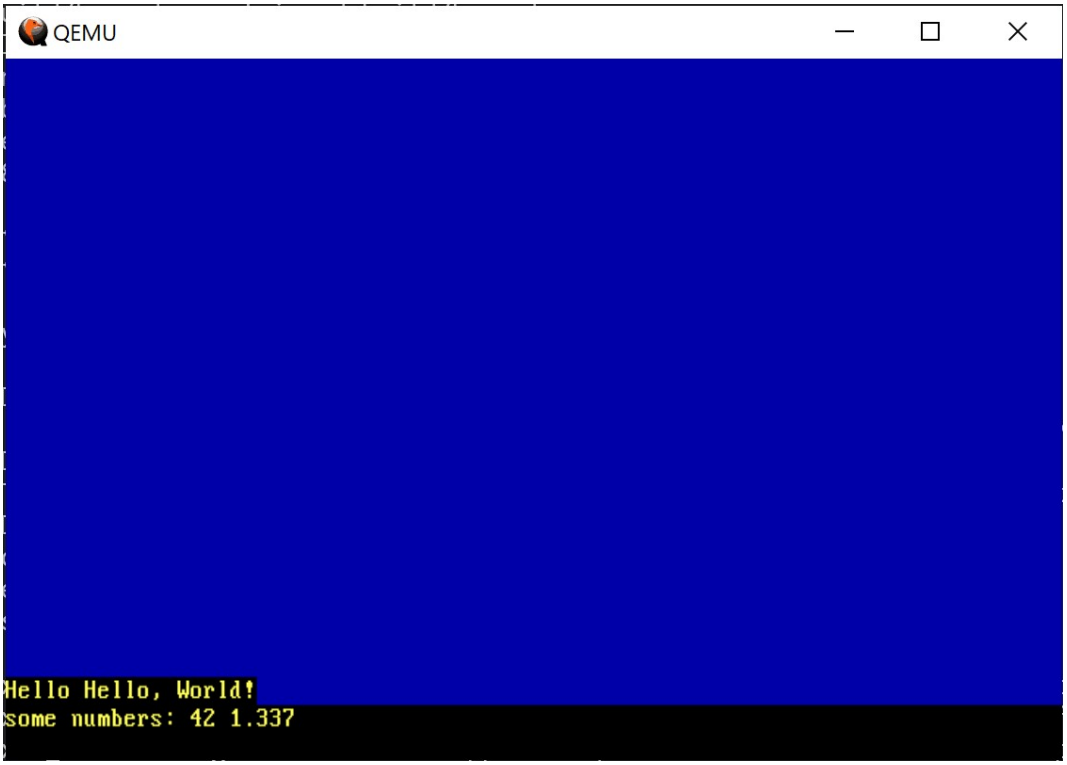


図 2　Multiboot2 を使ってブートに成功した。図 1 と変化を出すため、背景を青く塗りつぶした。

たりと、その生産性の高さは本当に驚異的だった。モダンで高級な言語機能を普通に使えるのも、やはりうれしかった。これは、OS を C++ で書いてみようという試みが目指しているものに近い何かを感じてしまう。もっとも、ベアメタルという観点では、C++ よりももっとお手軽に使えてしまえたけれど！ Rust はカーネル開発をもっと簡単に、そして楽しくしてくれそうだというのは間違いない。

今号の後半では、Multiboot2 を使って Rust 製カーネルのブートを行った。ほとんどがいつものブートプロセスなので、難しいところはなかったけれど、Rust はスタックが深くなるという点と、リンク周りの情報がまだ C 言語ほどはそろっていないという点は注意したい。この周りは少し、手探りで進む感じになるだろう。でも、これはきっとすぐに時間が解決してくれるだろう。Rust でベアメタル開発を行っている人は少なくないはずだ。

以上で、"Writing a (micro)kernel in Rust in 12 days"の 2nd day は終了だ。次なる 3 日目

では、一旦 Rust の世界からもベアメタルプログラムの世界からも離れ、L4 の世界に入ることになる。4 日目で Rusty L4 上で動かすための、L4Ka::Pistachio 用のサーバーを作るのだ。VGA のメモリ領域をたたいて、Hello, World! と表示させるだけのサーバーから始めるつもりだ。次回の Yabaitech.Tokyo で連載する（はず）だから、そのときにまたお会いしよう！

参考文献

[1] Jochen Liedtke. Improving IPC by kernel design. *14th ACM Symposium on Operating System Principles*, pages 175–188, 1993.

[2] ウィキペディアの執筆者 . *L4* マイクロカーネルファミリー . https://ja.wikipedia.org/wiki/L4 マイクロカーネルファミリー , 2019.

[3] Philipp Oppermann. *Writing an OS in Rust*. https://os.phil-opp.com, 2019.

[4] Free Software Foundation, Inc.. *Multiboot Specification version 0.6.96*. https://www.gnu.org/software/grub/manual/multiboot/multiboot.html, 2010.

[5] Free Software Foundation, Inc.. *https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html*. https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html, 2016.

[6] Free Software Foundation, Inc.. *Red Hat Enterprise Linux 4 Using ld, the Gnu Linker*. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Using_ld_the_GNU_Linker/index.html, 2004.

[7] csukuangfj. *This file shows the default linker script of`ld`, use`ld --verbose` to show it.*. https://gist.github.com/csukuangfj/c4bd4f406912850efcbedd2367ac5f33, 2019.

[8] OSDev Wiki Contributors. *Setting Up Long Mode - OSDev Wiki*. https://wiki.osdev.org/Setting_Up_Long_Mode, 2018.

[9] garasubo. *garasubo/erkos: A prototype embedded operating system written in Rust*. https://github.com/garasubo/erkos, 2019.

粘菌で計算がしたい！

wasabiz

1. はじめに

みなさんは「粘菌コンピュータ」というものをご存知でしょうか．粘菌コンピュータはその名が表す通り「粘菌」を用いてなんらかの「計算」を行う装置のことです．いわゆる自然計算と呼ばれる学術分野のなかで研究されているもので，電子や光子などのミクロな粒子が持つ量子性をうまく用いて高速な計算を行う**量子計算**，DNA や RNA などの生体分子を計算媒体として用いることで人工細胞や製薬などの応用を目指す DNA **計算**などと並列して，**粘菌計算**という分野として認知されています．

具体的な応用がわかりやすい量子計算や DNA 計算と比べると，粘菌コンピュータの目指す先はすこし漠然としています．実際，粘菌コンピュータを実際の計算機として使おうと考えている人は多分世の中にほとんど存在しないのではないかと思います．というのも，あとで詳しく説明しますが，粘菌コンピュータの計算媒体である「粘菌」というのはうねうねとゆっくり動く単細胞生物の一種で，そのうねうねという動きをうまく制御してなんらかの「計算」を行わせようというかなりクレイジーな研究が粘菌計算だからです．そもそも粘菌計算の研究と言っても研究者の数はかなり限られており，しかも論文を検索してみると大抵著者が日本人の研究グループだったりします．

とはいえ，研究者の数が限られているイコール全く認知されていないという訳ではありません．実際，「粘菌コンピュータ」という単語だけなら聞いたことがある人も多いのではないでしょうか．それもそのはずで，実は粘菌計算のクレイジーさは世界的にも認められており，なんと粘菌計算の研究の功績によって日本人の研究者たちが**イグノーベル賞**を受賞しています．しかも，2008 年と 2010 年の二回にわたってです．この受賞は日本でも大きくニュースになりました．当時を記憶している読者の中には「粘菌でパズル

を解く」とか「粘菌で東京の鉄道網を再現」といった表現に覚えがある方も多いと思います．より詳細には以下のような功績で粘菌計算の研究にイグノーベル賞が授与されています．(日本語版 Wikipedia からの抜粋)

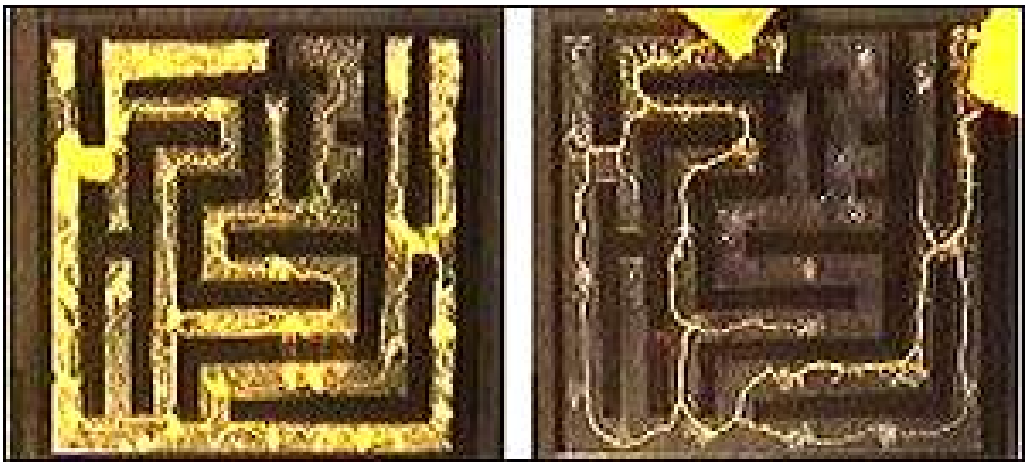
2008 年 認知科学賞	単細胞生物の真正粘菌に、パズルを解く能力があったことを発見したことに対して	中垣俊之 （北海道大学 / 理化学研究所） 小林亮（広島大学） 石黒章夫（東北大学） 手老篤史 （北海道大学 / Presto JST） 山田裕康 （名古屋大学 / 理化学研究所）
2010 年 交通計画賞	鉄道網など都市のインフラストラクチャー整備を行う際、真正粘菌を用いて、輸送効率に優れた最適なネットワークを設計する研究に対して。	中垣俊之 （公立はこだて未来大学） 小林亮（広島大学） 手老篤史 （科学技術振興機構さきがけプロジェクト） 高木清二（北海道大学） 三枝徹（北海道大学） 伊藤賢太郎（北海道大学） 弓木健嗣（広島大学）ら

粘菌の詳しい説明をする前に，粘菌計算で実現できる計算について簡単に説明しておきます．粘菌計算と一言で言っても大まかに二種類あります．一つ目は本物の粘菌を使って計算を行うもので，もう一つが粘菌の動きを数学的にモデル化してそのモデルを基にアルゴリズムを開発するものです．

(1) 本物の粘菌を使う

(2) 粘菌を数学的にモデル化する

一つ目の本物の粘菌を使う方は実際に粘菌がうにょうにょ動いているのを目でみながら実験ができるので粘菌を扱っている実感がして楽しいです．一方で，実際の粘菌を使う訳なので実験には時間と手間とお金がかかりますし，大規模なものになればなるほど実験がうまくいかなくなる可能性も高くなります．こちらの方法で実現されている有名なものがイグノーベル賞を受賞した「迷路の最短経路を発見する粘菌」や「理想的な鉄道網を発見する粘菌」などです．これらの「粘菌コンピュータ」は普通の人が「コンピュータ」と言った時に思い浮かべるようなものとは異なりますが，何らかの問題を自律的に解く装置であることには変わりないのでコンピュータと呼ばれています．



粘菌で迷路の最短経路を解いている写真．粘菌を全体に広げた後，始点と終点に餌を置くことで餌から離れた場所にいる足が徐々に引っ込んで行き始点と終点を結ぶ経路だけが残る．これが（大抵の場合）最短経路になっている，という研究．*14

二つ目の粘菌を数学的にモデル化する方法は，簡単に言えば黒板の上で仮想実験を行うというものです．粘菌の生体の内部構造を熱力学的にモデル化したり，あるいはそのモデルから着想を得た新しいアルゴリズムを開発したりします．この方法であれば数学的に厳密な方法で議論でき，しかも実験の手間に煩わされることもありませんが，「それ粘菌

14 http://news.bbc.co.uk/2/hi/sci/tech/944790.stm

を使う必要ある？」みたいな気分になってきます．たとえば「粘菌で SAT を解く」という研究なんかはこちらのアプローチです．

この記事では一つ目のアプローチの追試を目指していきます．とはいってもこの記事の範囲では粘菌を育てて可愛がるというところまでです．いくつか理由はあるのですが，迷路を解かせるために必要な程度の安定した大きさにするのが難しかったというのが一番の理由です．今回粘菌は研究室に置きっ放しで育てていたのですが研究室は夜には冷房を切ってしまうので梅雨とお盆の湿気と暑さでカビが大量発生してうまく粘菌を大きくできませんでした．なのでこの記事を読むにあたっては，粘菌のいる生活というものがどんな感じなのかというのを主に眺めていただくことにして，実際に粘菌で計算をする方法については紹介程度に止めることにしたいと思います．

2. 粘菌とは何なのか

ここまでで粘菌，粘菌と連呼してきましたが粘菌とは一体何なのでしょうか．粘菌は真核生物の一種で複数の形体を持ちます．その中でも有名なのが**変形体**という形体で，この形体は肉眼で見える非常に大きな姿をしており，ゆっくりと地面や壁をうにょうにょと餌を探して這いずり回ります．触った感じはネバネバとしており，ここから粘菌という名前がつけられているようです．また，変形体という特徴的な形体を持つことから粘菌という名前の代わりに**変形菌**と呼ばれることも多いようです．

ここまでの説明ですと非常にグロテスクな見た目を想像してしまうかと思いますが，実際には粘菌は多くの場合非常に綺麗な色と形をしており，見るものをうっとりさせます．粘菌の種類により見た目も形もさまざまなのですが，特に個人的に綺麗だと思ったものをここで引用しておきたいと思います．

タマツノホコリ*15

ヘビヌカホコリ*16

15 http://kuromedaka-saitama.cocolog-nifty.com/blog/2016/10/post-d7fb.html

16 http://furuhon-mayu.com/kinoko/hebinukahokori.htm

タマツノホコリ・クダホコリ・マメホコリ*17

ムラサキホコリ*18

17 https://www.1101.com/nenkin/2016-05-06.html

18 https://www.1101.com/nenkin/2016-05-06.html

キイロタマホコリ？ [*19]

このうち例えばムラサキホコリは変形体ではなく**子実体**と呼ばれる形体です．粘菌はその一生をさまざまな形体で過ごします．変形体として活動している粘菌はうにょにょと這いずり回りながらバクテリアなどを見つけて捕食して生きています．変形体は基本的には湿った暗い場所でしか生活できません．環境が悪くなると粘菌は変形体から子実体と呼ばれる小さなキノコのような形に変化します．そしてしばらくするとそのキノコのような形状の先端が壊れて中から多数の胞子が飛び出します．出てきた胞子はその後うまくいけば発芽し，**アメーバ体**に変わります．アメーバ体はバクテリアなどを捕食するのですが，ここで配偶子と出会うとそこから核分裂を繰り返して変形体へと変化します．この際，細胞分裂を行いません．つまり粘菌は単細胞生物でありその中に無数の核を持つという，少し変わった生物なのです．

粘菌自体は菌類っぽいのに動きまわるという不思議さ，森の中でひときわ目立つ鮮やかな色と奇抜な見た目により色々なもののモチーフになっています．おそらく中でも特に有名なのが漫画版の「風の谷のナウシカ」です．映画版の「風の谷のナウシカ」はスタジオジブリの代表作でもあり非常に有名ですが，その映画が実は宮崎駿自身の手による漫画の最初数巻分をアニメーションにしたものであるという事実はアニメや漫画好きの人以外には

19 https://blog.goo.ne.jp/tg660/e/97f630326eeab13946490f91406623a6

あまり知られていないかもしれません．その漫画版の中では映画版には登場しない「粘菌」なる生物が登場します．あんまりネタバレになるとよろしくないのですが，ナウシカの中で「科学者」たちが作り出して制御不能になった巨大な粘菌が登場します．変形体となった粘菌は人々の住む国中をうにょうにょと進みながら内部に取り込んでしまいます．

粘菌に飲み込まれる家々．

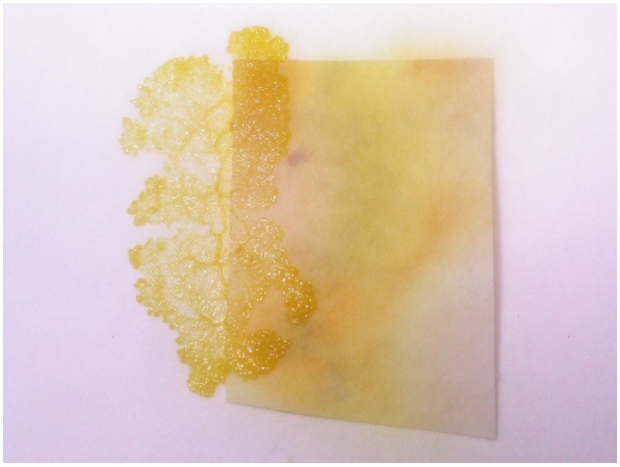
実は変形体という単語も出てくる.

空飛ぶ粘菌 (現実の粘菌はたぶん空を飛ばない)

ナウシカには他にも現実世界の生き物のオマージュがたくさん出てきますので興味のある方はぜひ読んでみてください.

3. 粘菌を育てる

実際の粘菌を使って実験を行うためには粘菌を飼う必要があるのですが，それはどれくらい難しいことなのでしょうか？実は，粘菌の飼育は**非常に簡単**です．実際，粘菌をペットとして飼っている人は多いようです．と言っても全ての種類の粘菌が簡単に飼育できるわけではありません．よくペットとして飼育されているのは**モジホコリ**と呼ばれる種類の粘菌です．



モジホコリの写真[*20]

モジホコリの変形体は黄色い見た目をしている粘菌です．嬉しいことに，モジホコリの変形体は**オートミール**を餌として食べます．基本的にはスーパーやコンビニで買ってきたオートミールで問題ありません．

20 https://ja.wikipedia.org/wiki/ 変形菌

スーパーで買ってきたオートミール．もちろん人間用．

実際にモジホコリを飼育するためには餌以外にも育てる環境を用意してやらねばなりません．と言っても，これもとても簡単です．まず粘菌を育てる培地ですが，これは普通のキッチンペーパーを濡らしたもので大丈夫です．ポイントは常に湿らせておくことですが，それもタッパーに入れておくだけで実現できます．(霧吹きは用意しておくと良いです．) あとは餌をあげたり，古くなった餌を取り除くためのピンセットもあればよいですが，別に割り箸でも代用できます．水はカルキとかはあんま気にしなくてもいいようです．

粘菌飼育セット (左から順に，キッチンペーパー，タッパー，ピンセット，霧吹き)

ある程度育ってきた場合は**寒天培地**で育てることも検討するといいでしょう．粘菌を育てる場合の寒天培地には特に栄養はいりません．スーパーで買ってきた寒天をそのまま栄養なしで固めるだけです．寒天の濃度ですが，手元にある寒天のパッケージの裏側を読むと「寒天一袋 (4g) を水 500mL に溶かして温めろ」と書いてあるので，デフォルトで作ると 0.8% の培地が出来上がります．通常粘菌の培養に用いられる寒天培地は 2% の濃度にするようなのですが，今回は寒天が勿体無いというのと，薄くても固まるならそれでいいだろうという判断で 0.8% 培地を作成していました．

また， タッパーよりもシャーレを用いた方が粘菌の状態が外から見えやすいというメリットがあります． これは毎度目視で確認しているのであればあまり手間が変わりませんが， たとえば定点カメラで粘菌を撮影しようとかし始めると少し問題になります． 大

抵のタッパーはポリプロピレン製で半透明だからです．加えて，タッパーの場合は蓋がぴったり閉まるようになっています．蓋をしない場合培地が急速に乾燥するので粘菌がすぐに弱ってしまいます．手元の環境では毎度蓋を完全に閉めていましたが，実はこれもよくない可能性があるので，タッパーの蓋の代わりにラップを軽くかける方がいいかもしれません．（たとえば長く放置したときに嫌気性の細菌が繁殖してしまうかもしれませんが，それが粘菌にとってどの程度影響があるのかはよくわかりません．）

ところで，粘菌の飼育ですが基本的には一日に一回餌やりと水やりをするだけで元気に生きてくれます．寒天培地は最初の用意が大変ですが，一度用意すれば水やりの手間も要らなくなります．古くなった餌は取り除いてやるほうが良いです．カビが生えてきた場合も取り除けるなら取り除いた方が良いです．キッチンペーパー培地の場合は細かいことは気にせずに元気なところだけ切り取って新しいキッチンペーパーの上に移し変えてやるのが良いようです．

4. 粘菌を手に入れる

ここまでで粘菌の育て方を簡単に説明してきましたが，まずは粘菌を手に入れないと話が始まりません．では粘菌はどうやって手に入れれば良いでしょうか．実は粘菌はその辺にいます．具体的には雑木林の中の地面や葉の裏，木の幹などです．あるいは，屋外にあるコンクリート壁や岩の垣根などの表面にいるという情報もあります．

地面に張り付いている粘菌っぽいなにか．

今回は粘菌を手に入れるために，まずは近所の雑木林を探しにいきました．結論からいうとそこで拾ってきた粘菌のような何かはうまく育てられませんでした．（結局本当に粘菌だったのかも不明です．実は単なる苔だったんじゃないかという気もします．）雨の日の翌日あたりに雑木林に行ってよく観察してみると木の根元には新しいキノコがいっぱい生えていることに気づきます．その要領でしっかり眺めていくとちょくちょく粘菌が見つかります．ただしそれらが育てられる粘菌，つまりモジホコリとは限りません．モジホコリは黄色の特徴的な見た目をしていますが，実はモジホコリの他にもモジホコリに非常によく似た黄色いススホコリという種がいるそうで，素人にはこれらの判断がつかないそうです．粘菌の同定は基本的に子実体を見て行うそうで，変形体を眺めて同定するのはほぼ無理なんだと思います．素直に専門家に頼みましょう．

東大本郷キャンパス内にある三四郎池という雑木林の中で拾ってきたキノコとか粘菌のような何か．結局なんなのかよくわからなかった．

さて，自分の足で見つけた粘菌を育てるという企みは上手く行かなかったので，代わりに別のところから手に入れることにします．最もお手軽な方法はインターネット通販でしょう．なんと実は，**粘菌（モジホコリ）はメルカリに売っています！**メルカリすごい．ちなみに出品価格は 2500 円でした．とても安いです．あいにく出品者はこの方ひとりなのですがメルカリで「モジホコリ」と検索するといわゆる「専用」ページが複数出てきます．粘菌というニッチな商品でもそれなりに売れているようで世の中の広さを実感します．

メルカリで出品されているモジホコリ

さて，メルカリで落札してから数日後，粘菌が到着しました．出品者は非常に丁寧な方で，オートミールに加えて育て方に関するノウハウをまとめた自作の資料まで送ってくださいました．感謝しかありません．

到着した粘菌．出品者の方の人の良さと粘菌愛が伺える．

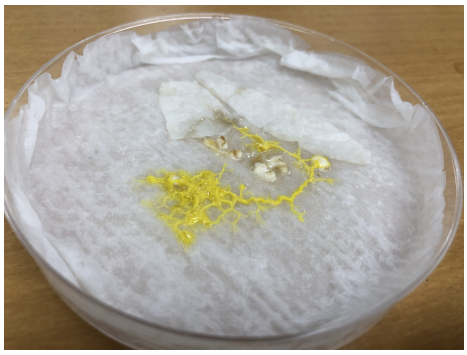
この元気な粘菌を頑張って育てていきます．

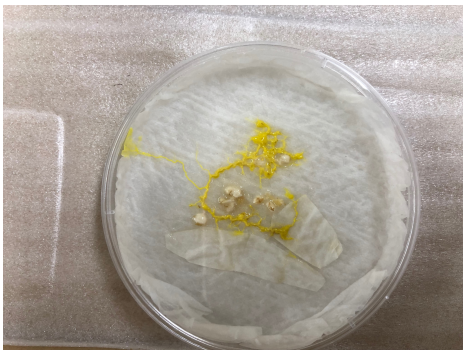
5. 粘菌飼育日記

粘菌の育つ過程を眺めていきます．結局計算といえるところまではたどり着かないのですが，粘菌を育てるということの雰囲気だけでもお見せします．

粘菌が到着してからとりあえず丸一日置いて成長を見ることにします．粘菌は光を嫌がるため，普段は段ボール箱の中に入れておきます．なお，粘菌の変形体を飢餓状態に置いたのち光を当てると子実体に変化するそうです．これを後々実験してみたのですがうまくいかず，子実体を作ることなく枯れて（？）しまいました．なのでこの記事では写真とかは特に載せません．

さて，丸一日置いた翌日の粘菌の状態が以下の通りです．





初日の粘菌（左）と丸一日置いた後の粘菌（右）

左に向かって大きく足が伸びていることがみて取れます．元気に活動しているようです．6 月の末で最低気温が 22 度前後だったので気温がちょうどよかったのも幸いしたと思います．ここからしばらく旅行に出かけたためしばらく粘菌をほったらかしにしました．

網走でとった写真．星が綺麗．

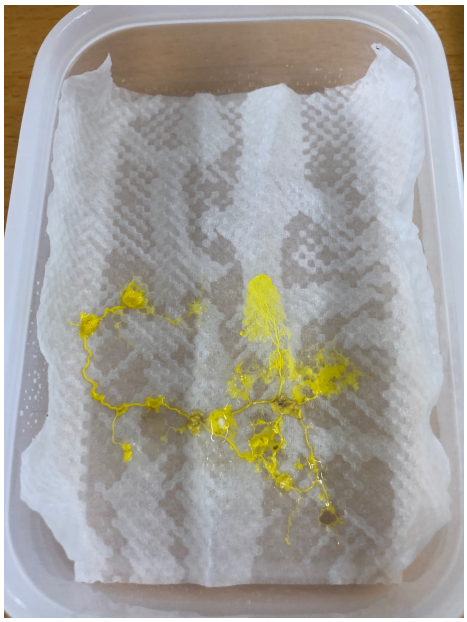
旅行から帰ってくると見事にキッチンペーパーにカビが生えていました． ということで元気なところだけを切り取って別のキッチンペーパー培地に移植．

カビが生えた培地（１枚目）を切り取って新しいキッチンペーパー培地に移植（２枚目）.

粘菌は単細胞生物ですが，適当に切り取っても残りの部分は普通に生きています．というより，真っ二つに切ると二つに分裂します．これを利用して株分けを行っていき，実験の規模を大きくしていきます.

株分けを行う．少しずつ大きくなっていく．

ちなみに粘菌の移動速度ですが，肉眼で見えるほどには速くありません．ただし一時間ごとに見ていくと移動していることがはっきり分かる程度には速いです．たとえば，同じ日の 13 時と 17 時半にとった写真二枚を比較すると，その 4 時間半の間にかなり大きく移動していることが分かるでしょう．

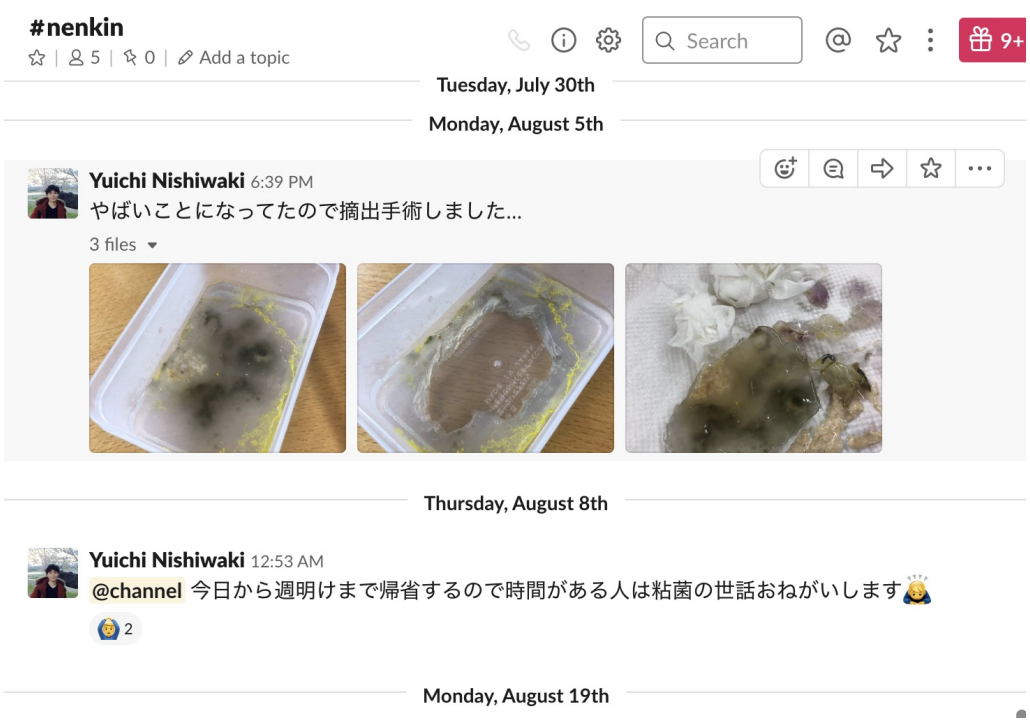


粘菌の移動速度．一つ目が 13 時 00 分に撮ったもの．二つ目が同じ日の 17 時 30 分に撮ったもの．4 時間半でもかなり大きく移動している．

この辺りから粘菌の飼育に慣れてきたので，タッパーの数と大きさをどんどん大きくしていきます．

増えるタッパー．（タッパー５つとシャーレ１つ）

この辺りで 8 月に入り異常に暑くなりました．最低気温でも 25 度を超えるようになり，冷房の効かない部屋に長時間放置されたタッパーには大量のカビが生えてしまいます．



Slack の粘菌チャンネルでカビの報告をする人.

その後さらに研究室全体がお盆休みに入りほとんどの人が粘菌の世話をできなくなった結果，カビの状況が悪化しました．結果的に，カビの魔の手から逃げようとする粘菌が培地を捨ててタッパーの蓋（天井）を目指す事態に ....

汚染された大地を捨てて空を目指す粘菌たち．（寒天培地はカビで真っ黒）

Yabaitech.tokyo の原稿の締め切りも迫るなか，カビの大繁殖によって迷路を解くってレヴェルじゃなくなってしまったので迷路の実験は一旦中止． 今後やろうと思っていた次号以降に延期することにし，ひとまず安定的な粘菌の飼育に専念することにしました．

6. 粘菌の魅力

粘菌を迎え入れてからおおよそ 2 ヶ月面倒を見てきましたが粘菌には独特の魅力があります． ペットというには動きが遅すぎ， 観葉植物として見るには少々華が足りない粘菌ですが， 裏を返せば， ペットほどの世話が必要なくかつゆっくりながらも動きがあり見ていて面白い生き物でもあります．

世話に関して 1 つ補足をしておくと， 実は粘菌は冷蔵庫に入れて保存することができ

るそうです．粘菌をある程度飢餓状態に置いた後冷たい場所で徐々に乾燥させると粘菌は変形体から**休眠体**と呼ばれる新たな形体に変わります．この形体ではもはや水をやる必要すらなく，冷蔵庫に安置することで 1 年から 2 年程度保存することが可能とのことです．さらに休眠体を変形体戻すときは水をやるだけで戻るというお手軽さです．なので，粘菌と遊びつくして世話するのが面倒になったという時には休眠体にしておき，また遊びたくなったらいつでも元に戻すことができます．実は，粘菌を手に入れてから休眠体にする実験も行ったのですが上手く水で戻すことができず，失敗してしまいました．なので，粘菌を保存するためにはある程度コツが必要なのだと思いますが，以上のことは様々な文献に記述されているので私の手際が悪かっただけかもしれません．

粘菌がペットとして独特な点として，ちょっとずつ切り取って実験に使えるということがあります．基本的に環境さえ整えばいくらでも大きくなっていくので，一部だけ切り取って遊ぶだけ遊んでも残りの部分にはダメージはなく，元気に大きくなっていきます．（直観的には）生きているのか生きていないのかもよくわからないので，生物実験の中でもそんなに罪悪感がない方ではないかと思います．

粘菌は水と餌さえあればいくらでも大きくはなるのですが，与え方にもコツがあります．あまり多くの餌を一度に与えすぎるとある程度以上食べた後はそのまま残りに手をつけずにそっぽを向いてしまいます．オートミールの種類によっても好き嫌いがあるようで，メルカリでいただいたオートミールは安定して食べてくれているようです．他にもオートミールの配置の仕方など細かいところで工夫するところが多く，単に見ていて楽しいだけでなく粘菌をどんどん元気にするための工夫ができる点も大きな魅力です．

あとこれはペット全体に言えることだと思いますが，ふと作業に飽きた時にじーっとながめる対象がいるというのはなかなか精神的に良いものです．数十秒程度では粘菌の形に変化はないとわかっていてもなんとなく眺めてしまいます．（一方で一時間おきに見れば粘菌が餌を探して動き回っていることがよくわかります．）ある程度大きく育っていればしばらく世話を忘れてしまっても全然元気に生きています．小さい変な生き物ですが，分かりやすくて元気な生き物でもあります．科学的にも興味深く，ペットとしても可愛い粘菌，興味が出た方はぜひ一度，育ててみてはいかがでしょうか？

真矛盾主義入門

zeptometer

1. 序文

「矛盾」という言葉が中国の古典から来ていることはよく知られていますよね。「この盾はどんな攻撃も防ぐぞ」「この矛は全てのものを貫くんだ」なんて調子のいいことをぬかしていた商人が「じゃあその矛でその盾を突いたらどうなるんだ」と真っ当な指摘をされて何も言えなくなってしまったという話です。

この話でもそうなのですが、思考や議論に矛盾を生じることを我々はよしとしません。結論が矛盾しているならそれを導く過程や前提のどこかが間違っていると考えるのです。矛盾があってはならないという考えは論理体系の形式化においても最も基本的なアイデアの一つになっています。

しかしながらこの無矛盾の原則が必ずしも成り立たないと考える人たちがいます。彼らによると矛盾の中には「真なる矛盾」、真でありかつ偽であるような命題が存在するというのです。このような立場を**真矛盾主義** (dialetheism[*21]) と呼びます。この記事は真矛盾主義の動機付けやその形式化について紹介します。

注意：この記事の 3 章以降は古典論理の意味論とシーケント計算の基礎的な知識を前提としています。

2. 「何故」真矛盾主義か

"Logic: A very short introduction"[1] という論理学の入門書があります。この本のい

21 dialetheism という語は di-aletheia(=「二つの真理」) という造語から来ています。

い所は新しい論理を紹介する時に「自然言語にこういう推論や概念があって、これをうまく表現する論理体系が欲しい」という感じで何故そういう論理を考えたいのかという動機付けをわかりやすく説明してくれている点です。

先程の章で「真でも偽でもあるような文が存在する」という主張を聞いた時に、最初に出る質問はきっと「何故そんなややこしい状況を考えたいのだろうか」になるでしょう。それに対する説明をいくつか紹介します。

2.1. 矛盾する法律

一般的に法体系というのは巨大で、しばしばその中に矛盾するような条文があるというのはいかにもありそうなことです。[2] からの引用ですが、例えばオーストラリアの選挙法[*22]にこんな条文があったとしましょう。

- 全てのアボリジニは投票権を持たない
- 全ての土地所有者は投票権を持つ

多分この法律ができた当時はアボリジニが土地を持つなんてのは考えられなかったんでしょうねえ (例のために書いてるだけで全く架空の条文です。いいですか)。先住民の差別が撤廃されていく過程でアボリジニの土地所有者がでてくるとこれらの条文が矛盾することになり困ったことになってしまいます。

これには色々な対策が考えられます。法律が矛盾を含まないような仕組みを導入するのはどうでしょう。例えば「先に書かれている条文を優先する」というルールを導入すればとりあえず先の条文の矛盾は回避されます。しかし他の部分で矛盾が生じないという保証にはなりません。矛盾が発見されしだいすぐ修正するようにするという対策もあります。しかし修正されるまではその法律は矛盾したままになりますし、矛盾が発見されていなかったとしてもその法律が矛盾をはらんでいることには変わりありません。

結局のところ、法律に矛盾が生じる可能性を受け入れる必要がありそうです。そのような場合に「矛盾があるからこの法律は無意味だ」という結論にしていては司法が立ちいかなくなってしまいます。代わりにその部分が矛盾していることを認めた上で妥当な

22 オーストラリアなのは引用元の著者がオーストラリア人だからです。日本の事情に即した条文つくりたかったけど結局いいのが思いつかなかった。

推論を重ねて判断を下していくことになるでしょう。このような立場は真矛盾主義の主張に合致します。

2.2. 自己言及のパラドックス

もう一つ真矛盾主義によって綺麗に説明できる概念があります。**自己言及のパラドックス**です。

(1) : (1) は偽である

こんな感じの文です。この文自体は**嘘つきのパラドックス**と呼ばれています。この文は真なのでしょうか、偽なのでしょうか。それぞれの場合で考えてみましょう。

- (1) が真 → (1) より「(1) は偽」なので (1) は偽
- (1) が偽 → (1) の「(1) は偽」に一致するので (1) は真

どちらにしても前提の反対の結論がでてきて矛盾が生じてしまいますね。この嘘つきのパラドックスを最初に言い出したのはエウブリデスという古代ギリシアの哲学者らしく、歴史のあるパラドックスです。そしてその長い歴史にも関わらず、このパラドックスに対する広く認められた解答は未だに存在しません。

解答の一つとしては「自己言及のパラドックスは文として成り立っていない」というものが考えられます。例えば集合論におけるラッセルのパラドックスも自己言及のパラドックスの一種ですが、その解答としての公理的集合論はこのようなパラドックスをそもそも記述できないようにすることで矛盾を回避していると言えます。

真矛盾主義は嘘つきのパラドックスを含む自己言及のパラドックスに対し別の解答を提示します。すなわち「(1) は真でありかつ偽である」という解釈です。(1) が真かつ偽であるとすれば、(1) は真なので「(1) は偽」となりますが実際に (1) は偽でもあります。一方 (1) は偽なので「(1) は偽」に一致して (1) は真となりますが、(1) は真かつ偽なので問題ありません。屁理屈に聞こえるかもしれませんが、しかし辻褄は合っています。

3. First Degree Entailment の意味論

このような真矛盾主義という考え方があって、それを形式的な論理体系に落としこみ

たいというのはごく自然な考えですよね。そのような論理体系として First Degree Entailment (FDE)[3] を紹介します。

まず記号の定義をしていきましょう。FDE の命題 $A, B, \ldots$ は**原子命題** p、**否定** $\neg A$、**連言** $A \wedge B$、**選言** $A \vee B$ から成ります。気持ちの説明をすると原子命題は何らかの知識の対象 (「猫はかわいい」とか「パリはアメリカの首都だ」とか) を抽象化したもので $p, q, \ldots$ と書き表します。否定 $\neg A$ は「A ではない」、連言 $A \wedge B$ は「A かつ B である」、選言 $A \vee B$ は「A または B である」という意味に解釈できます。

古典論理だと命題は真であるか偽であるかのどちらかですよね。FDE はこれを一般化してある命題が「真であるかどうか」と「偽であるかどうか」を独立した性質として解釈します。

定義 1 (原子命題の解釈)　原子命題の**解釈** $\rightarrow$ は原子命題と真偽値 $(= \{0, 1\})$ の二項関係である。 ◇

$p \rightarrow 1$ は「p は真である」、$p \rightarrow 0$ は「p は偽である」という意味になります。ここである p において $p \rightarrow 1$ かつ $p \rightarrow 0$ である、あるいは $p \rightarrow 1$ でも $p \rightarrow 0$ でもないという状況がありうるのがポイントです。 この原子命題の解釈を拡張して一般の命題の解釈を与えましょう。

定義 2 (命題の解釈)　命題の解釈は命題と真偽値の二項関係である。 原子命題の解釈 $\rightarrow$ が与えられた時、これを以下のように拡張して命題の解釈とする。

$$
\begin{aligned}
A \wedge B \rightarrow 1 \quad &\text{iff} \quad A \rightarrow 1 \quad \text{and} \quad B \rightarrow 1 \\
A \wedge B \rightarrow 0 \quad &\text{iff} \quad A \rightarrow 0 \quad \text{or} \quad B \rightarrow 0
\end{aligned}
$$

$$
\begin{aligned}
A \vee B \rightarrow 1 \quad &\text{iff} \quad A \rightarrow 1 \quad \text{or} \quad B \rightarrow 1 \\
A \vee B \rightarrow 0 \quad &\text{iff} \quad A \rightarrow 0 \quad \text{and} \quad B \rightarrow 0
\end{aligned}
$$

$$
\begin{aligned}
\neg A \rightarrow 1 \quad &\text{iff} \quad A \rightarrow 0 \\
\neg A \rightarrow 0 \quad &\text{iff} \quad A \rightarrow 1
\end{aligned}
$$

◇

例えば連言の例だと「$A \wedge B$ が真」であるとは「A が真でかつ B が真」であるということ

で、一方で「$A \wedge B$ が偽」であるとは「A か B の少なくとも一方が偽である」ということになります。否定の定義は真と偽をひっくり返す定義になっていてわかりやすいですね。

真と偽が別の性質になっているということは $A \to 1$ であることと $A \to 0$ でないことが独立した事象であることを意味しています。こういった議論をしやすくするために以下の記法を導入します。

記法 3　ある解釈 $\to$ が与えられた時、「$A \to i$ ではない」ことを「$A \nrightarrow i$」と表記する。 ◇

さて、ある命題 A が与えられた時に我々は以下の 4 つの可能性を考えることができます。

- $A \to 1$ and $A \nrightarrow 0$ (A は真である)
- $A \nrightarrow 1$ and $A \to 0$ (A は偽である)
- $A \to 1$ and $A \to 0$ (A は真かつ偽である)
- $A \nrightarrow 1$ and $A \nrightarrow 0$ (A は真でも偽でもない)

ということはこれらにそれぞれ論理値を割りあてて 4 値論理とみなすことができます。ここでは t(true), f(false), b(both), n(neither) と呼ぶことにしましょう。また、以下のような記法を導入します。

定義 4 (4 値解釈)　FDE の **4 値解釈**とは FDE の命題と $\{t, f, b, n\}$ の二項関係である。FDE の解釈 $\to$ が与えられた時、以下のように FDE の 4 値解釈 $\to_4$ を定義できる。

$$
\begin{aligned}
A \to_4 t \quad & \text{iff} \quad A \to 1 \quad \text{and} \quad A \nrightarrow 0 \\
A \to_4 f \quad & \text{iff} \quad A \nrightarrow 1 \quad \text{and} \quad A \to 0 \\
A \to_4 b \quad & \text{iff} \quad A \to 1 \quad \text{and} \quad A \to 0 \\
A \to_4 n \quad & \text{iff} \quad A \nrightarrow 1 \quad \text{and} \quad A \nrightarrow 0
\end{aligned}
$$

◇

ここでは間接的に 4 値論理のようなものをつくっているだけですが、直接 4 値論理として意味論を与えることももちろん可能です。詳しくは [2, 3] を参照してください。

これらの論理結合子に関する定義はそれぞれの命題が t か f である場合には古典論理と同じふるまいをすることが確認できます。そういった意味で FDE は古典論理を拡張したものだと言えそうです。しかしそうでない状況の場合には面白い結果が得られます。例として排中律 $A \vee \neg A$ がどのように解釈されうるかを考えてみましょう。$A \rightarrow_4 n$ である時、¬(と 4 値解釈) の定義より $\neg A \rightarrow_4 n$ になり、次に $\vee$ の定義より $A \vee \neg A \rightarrow_4 n$ であることが言えます。逆に $A \rightarrow_4 b$ であるような状況を考えると、¬ の定義より $\neg A \rightarrow_4 b$ で、さらに $\vee$ の定義より $A \vee \neg A \rightarrow_4 b$ であることが導けます。これは FDE において排中律が「真でない」あるいは「偽である」状況がありうることを意味しています。同様のことが無矛盾則 $\neg(A \wedge \neg A)$ でも成り立ちます。

4. 双シーケント計算による推論体系

先程の話は FDE の意味論の側の話でしたが、FDE を論理体系として成立させるためにはその意味論に対応する推論体系も必要です。Belnap による FDE の元論文 [3] でもそういった推論規則を提示していますが、この記事では Bochman による Biconsequence Relation を用いた推論体系 [4] をベースにした**双シーケント計算**による推論体系を紹介します。

4.1. 双シーケント計算

双シーケント計算は古典論理のシーケント計算[*23]を拡張したものです。命題の有限列を $a, b, \ldots$ と表記することにしましょう。古典論理のシーケント計算では、

$$a \vdash b$$

と表記して「a の命題が全て真でかつ b の命題が偽であることはない」[*24]ことを主張するものでした。これを**シーケント**と呼びます。FDE の双シーケント計算ではこれを拡張して、以下のような形の**双シーケント**を用います。

$$a : b \vdash c : d$$

---

23 古典論理のシーケント計算はそれこそ古典なので検索したら山ほど資料が出てきます。詳細についてはそれらをご参照ください :P

24 これは「a の命題が全て真である時、b の命題のうち少なくとも一つが真である」と同値で、多分こっちの方が馴染みのある定義だと思います。この記事では後でする説明の都合上このように言い代えています。

古典論理のシーケントでは 2 つパートがあったのに対し、こちらでは 4 つになっています。この双シーケントは「a の全てが真で、b の全てが偽で、c の全てが真でなく、かつ d の全てが偽でないということはない」ことを主張します。真と偽が別々の性質になったのでシーケントの持つ情報が倍になったわけです。これは意味論の言葉を使うと以下のような定義で表わされます。

定義 5 (妥当な双シーケント)　ある双シーケント $a : b \vdash c : d$ が FDE の解釈 $\to$ に対して**妥当**であるとは、「全ての $A \in a$ について $A \to 1$、かつ全ての $B \in b$ について $B \to 0$、全ての $C \in c$ について $C \not\to 1$、全ての $D \in d$ について $D \not\to 0$」**ではない**ことである。また、ある双シーケントが任意の解釈に対して妥当であるとき、単にそのシーケントを妥当であるという。 ◇

うーん長い。でも言ってることは単純なので最後の「ではない」にさえ注意してくれれば多分問題ないです。

以降特に新しい推論体系が出ることもないので、古典論理のシーケント計算と FDE の双シーケント計算をそれぞれ単にシーケント計算と双シーケント計算と呼ぶことにしましょう。シーケント計算と同じく、双シーケント計算では様々な規則を適用することで推論を行います。規則には双シーケントの性質を表す構造規則と論理結合子の意味を規定する規則に分かれます。

双シーケント計算の構造規則はシーケント計算の構造規則を拡張したものになっています。図 1 が構造規則の一覧です。各規則名の中の P と N は Positive と Negative の略で双シーケントの真の部分か偽の部分かに対応します。また、L と R は双シーケントの左側 (Left) か右 (Right) かを表しており、以下のように双シーケントのそれぞれの部分に対応します。

$$(\mathrm{PL}) : (\mathrm{NL}) \vdash (\mathrm{PR}) : (\mathrm{NR})$$

これらの構造規則のもっともらしさは双シーケントについての直観によって正当化されます。双シーケントのお気持ちを振り返ると、$a : b \vdash c : d$ という双シーケントは「a の全てが真で、b の全てが偽で、c の全てが真でなく、かつ d の全てが偽でないこと」はありえないということを表すのでした。この「a の全てが真で、...」の部分を直接書くと長った

$$\text{(Weakening)}\ \frac{a : b \vdash c : d}{a,a' : b,b' \vdash c,c' : d,d'}$$

$$\text{(Exchange PL)}\ \frac{a,A,B,a' : b \vdash c : d}{a,B,A,a' : b \vdash c : d} \qquad \text{(Exchange NL)}\ \frac{a : b,A,B,b' \vdash c : d}{a : b,B,A,b' \vdash c : d}$$

$$\text{(Exchange PR)}\ \frac{a : b \vdash c,A,B,c' : d}{a : b \vdash c,B,A,c' : d} \qquad \text{(Exchange NR)}\ \frac{a : b \vdash c : d,A,B,d'}{a : b \vdash c : d,B,A,d'}$$

$$\text{(Contraction PL)}\ \frac{a,A,A : b \vdash c : d}{a,A : b \vdash c : d} \qquad \text{(Contraction NL)}\ \frac{a : b,A,A \vdash c : d}{a : b,A \vdash c : d}$$

$$\text{(Contraction PR)}\ \frac{a : b \vdash c,A,A : d}{a : b \vdash c,A : d} \qquad \text{(Contraction NR)}\ \frac{a : b \vdash c : d,A,A}{a : b \vdash c : d,A}$$

$$\text{(Reflexivity P)}\ \frac{}{A : \vdash A :} \qquad \text{(Reflexivity N)}\ \frac{}{: A \vdash : A}$$

$$\text{(Cut P)}\ \frac{a : b \vdash A,c : d \qquad A,a : b \vdash c : d}{a : b \vdash c : d}$$

$$\text{(Cut N)}\ \frac{a : b \vdash c : A,d \qquad a : A,b \vdash c : d}{a : b \vdash c : d}$$

図 1　双シーケント計算の構造規則

らしくて嫌なので以降「※」と書くことにしてしまいましょう。

さて、「※」がありえないのであれば、「A が真でありかつ ※」のように水増しをしても変わらずありえないはずです。この直観に対応するのが (Weakening) の規則です。また、「※」のうち a の中の命題の順番を入れ替えても「※」がありえないことは変わりませんし、a の中にいくつ同じ命題が出てきたところでその命題が 1 つしかない時と状況は変わりません。これらの直観にはそれぞれ各 Exchange 規則と各 Contraction 規則が対応しています。

少し話が逸れますが、双シーケントにおける「ありえない」状況の最小単位は何になるでしょうか。3 節で説明した意味論を直観の助けにして考えてみましょう。例えば「A が真でかつ A が真でない」という状況はありえないはずですよね、ある解釈 $\to$ が与えられた

ら $A \to 1$ か $A \nrightarrow 1$ のどちらかは必ず成り立ちますから。同様に「A が偽でかつ A が偽でない」という状況もありえないはずです。これらの直観を構造規則として表現したものがそれぞれ (Reflexivity P) と (Reflexivity N) です。一方で FDE では「A が真かつ偽である」という状況や「A は真でも偽でもない」という状況がありうるのは 3 節で話した通りです。そのため以下のようなものは構造規則には入らないことになります。

$$\text{(Reflexivity L)}\ \frac{}{A : A \vdash\ :} \qquad \text{(Reflexivity R)}\ \frac{}{:\ \vdash A : A}$$

(Cut) は違う場所にある同じ命題を打ち消す構造規則です。例えば「A が真でありかつ ※」ではないことと「A が真ではなくかつ ※」ではないことの両方がわかっていれば、A が真であろうとなかろうと「※」ではないことがわかりますよね。この直観を構造規則にしたものが (Cut P) です。この真を偽にひっくり返せば (Cut N) になります。

4.2. シーケントから双シーケントへ

先ほどから何度か「FDE は古典論理の拡張である」という主張をしてきましたが、そうであればシーケントを双シーケントに埋め込むことができるはずです。これをどう実現するか考えてみましょう。シーケントは「真」と「偽」の 2 つのパートに分かれていたところが、双シーケントではこれらが細分化されて「真である」「偽である」「真でない」「偽でない」の 4 つのパートに分かれているのでした。このうち「真」は「真である」と「偽でない」、「偽」は「真でない」と「偽である」に対応づけられそうですね。

結論から言ってしまうとシーケントから双シーケントへ埋め込むにあたって以下の 2 通りの対応づけが健全な埋め込みを与えます[*25]。

- (真)⇒(真である), (偽)⇒(真でない)
- (真)⇒(偽でない), (偽)⇒(偽である)

これをシーケントから双シーケントへの対応づけとして表現すると以下のようになります。

- (Positive Embedding) $a \vdash b \Rightarrow a :\ \vdash b :$

25 ここでの健全性とは妥当性を保つ埋め込みであることを意味します。多分成り立つけど証明はないです。間違ってたらごめん .....

- (Negative Embedding) $a \vdash b \Rightarrow \ : b \vdash \ : a$

このようにシーケントを双シーケントに埋め込むにあたって 2 通りの解釈を考えることができます。

4.3. FDE の論理結合子に関する規則

次に論理結合子に関する規則を見ていきます。先ほどシーケントから双シーケントへの埋め込みについて議論したところなので、発展としてシーケント計算の規則を用いて双シーケント計算の規則をつくってみましょう。まずはシーケント計算の連言に関する規則の一つ (∧L) を変換します。

$$(\wedge\mathrm{L})\ \frac{a, A, B \vdash b}{a, A \wedge B \vdash b}$$

まず規則に出てくるシーケントを Positive と Negative のそれぞれの埋め込みで変換すると以下のものが得られます。

$$(\wedge\mathrm{PL?})\ \frac{a, A, B : \ \vdash b :}{a, A \wedge B : \ \vdash b :} \qquad (\wedge\mathrm{NR?})\ \frac{: b \vdash \ : a, A, B}{: b \vdash \ : a, A \wedge B}$$

このままだと規則中の双シーケントに空の部分があってよくないので適当に埋めちゃいましょう。

$$(\wedge\mathrm{PL})\ \frac{a, A, B : b \vdash c : d}{a, A \wedge B : b \vdash c : d} \qquad (\wedge\mathrm{NR})\ \frac{a : b \vdash c : d, A, B}{a : b \vdash c : d, A \wedge B}$$

うんうん、なかなかいい感じですね。実際にこの規則が全節の FDE の意味論に対して健全であることは容易に確認できるはずです。

この調子で選言、連言、否定について双シーケント計算での規則を作ることにしましょう。

- ∨ に関する規則

$$(\vee\mathrm{PL})\ \frac{a, A : b \vdash c : d \qquad a, B : b \vdash c : d}{a, A \vee B : b \vdash c : d} \qquad (\vee\mathrm{PR})\ \frac{a : b \vdash c, A, B : d}{a : b \vdash c, A \vee B : d}$$

$$(\vee\mathrm{NL})\ \frac{a : b, A, B \vdash c : d}{a : b, A \vee B \vdash c : d} \qquad (\vee\mathrm{NR})\ \frac{a : b \vdash c : d, A \qquad a : b \vdash c : d, B}{a : b \vdash c : d, A \vee B}$$

- $\wedge$ に関する規則

$$(\wedge\mathrm{PL})\ \frac{a, A, B : b \vdash c : d}{a, A \wedge B : b \vdash c : d} \qquad (\wedge\mathrm{PR})\ \frac{a : b \vdash c, A : d \qquad a : b \vdash c, B : d}{a : b \vdash c, A \wedge B : d}$$

$$(\wedge\mathrm{NL})\ \frac{a : b, A \vdash c : d \qquad a : b, B \vdash c : d}{a : b, A \wedge B \vdash c : d} \qquad (\wedge\mathrm{NR})\ \frac{a : b \vdash c : d, A, B}{a : b \vdash c : d, A \wedge B}$$

- $\neg$ に関する規則

$$(\neg\mathrm{PL?})\ \frac{a : b \vdash c, A : d}{a, \neg A : b \vdash c : d} \qquad (\neg\mathrm{PR?})\ \frac{a, A : b \vdash c : d}{a : b \vdash c, \neg A : d}$$

$$(\neg\mathrm{NL?})\ \frac{a : b \vdash c : d, A}{a : b, \neg A \vdash c : d} \qquad (\neg\mathrm{NR?})\ \frac{a : b, A \vdash c : d}{a : b \vdash c : d, \neg A}$$

ちょっと待ってください。どうも最後の否定に関する規則は意味論にうまく対応していないようです。例えば以下のような推論を考えると、$A \to_4 b$ であるような $\to$ において妥当な前提から妥当でない結論が導かれてしまいます。

$$(\neg\mathrm{PL})\ \frac{: \vdash A :}{\neg A : \vdash :}$$

何かを間違ったのか？というとそういうわけではありません。ネタバラシをすると、こうやって得られる否定は実は 3 節で定義した否定とは別物なのです。先ほど古典論理の否定から得た否定は local negation と呼ばれるもので、以降は区別のために $\neg A$ ではなく $\sim A$ と書くことにします。local negation の解釈は以下のように定義されます。

$$\sim A \to 1 \quad \text{iff} \quad A \nrightarrow 1$$
$$\sim A \to 0 \quad \text{iff} \quad A \nrightarrow 0$$

対して 3 節で定義した否定を switching negation と呼びます。local negation は真であるかないか偽であるかないかを裏返すものであるのに対し、switching negation を真と偽を入れ替えるものというわけです。それぞれの双シーケント計算における定義は以下のようになります。

- ¬ (switching negation) に関する規則

$$(\neg\text{PL})\ \frac{a : b, A \vdash c : d}{a, \neg A : b \vdash c : d} \qquad (\neg\text{PR})\ \frac{a : b \vdash c : d, A}{a : b \vdash c, \neg A : d}$$

$$(\neg\text{NL})\ \frac{a, A : b \vdash c : d}{a : b, \neg A \vdash c : d} \qquad (\neg\text{NR})\ \frac{a : b \vdash c, A : d}{a : b \vdash c : d, \neg A}$$

- ~ (local negation) に関する規則

$$(\sim\text{PL})\ \frac{a : b \vdash c, A : d}{a, \sim A : b \vdash c : d} \qquad (\sim\text{PR})\ \frac{a, A : b \vdash c : d}{a : b \vdash c, \sim A : d}$$

$$(\sim\text{NL})\ \frac{a : b \vdash c : d, A}{a : b, \sim A \vdash c : d} \qquad (\sim\text{NR})\ \frac{a : b, A \vdash c : d}{a : b \vdash c : d, \sim A}$$

これらは古典的な設定 (= 全ての命題が t か f である) 場合には同じように振舞います。なので古典論理における否定が local negation と switching negation に分裂したと解釈することもできます。ところで面白いことにいくつかの文献 (例えば [2] とか) では FDE における否定の論理結合子として switching negation のみを提示しており、古典論理の否定を素直に拡張して得られるはずの local negation には言及していません (3 節でもそのような流れになっているのはそういうことです)。紙面上の都合もあるのでしょうが、FDE の設定で初めて考えることのできる switching negation こそが「否定」として適切な概念だという考えが根底にあるのではないかと思われます。

4.4. 基本的な性質

これまで紹介した双シーケント計算の体系ですが、重要な性質として以下のものが成り立つと思われます。思われると書くのは証明したわけではないのと、元論文で紹介されていた体系に多少手を加えている結果として元論文で言及されていた性質がこの記事で紹介している双シーケント計算に成り立つかが不明であるためです。この体系が適切に定義されていたら成り立つはずの性質、くらいに思っていただければ。

定理 6 (カット除去定理)　双シーケント計算による推論があった場合、同じ双シーケントを *(Cut P)* 及び *(Cut N)* を除いた規則で証明できる。 ◇

定理 7 (Reflexivity の原子命題への制限)　双シーケント計算による推論があった場合、同じ双シーケントを *(Reflexivity P)* と *(Reflexivity N)* を除き代わりに以下の規則を追加したもので証明できる。

$$\textit{(Reflexivity' P)}\ \frac{}{p: \vdash p:} \qquad \textit{(Reflexivity' N)}\ \frac{}{:p \vdash :p}$$

◇

定理 8 (健全性)　双シーケント計算の推論規則によって証明された双シーケントは妥当である。 ◇

定理 9 (完全性)　妥当な双シーケントは双シーケント計算の推論規則によって証明できる。 ◇

4.5. 部分構造論理としての真矛盾主義

ここで少し 4.1 節における Reflexivity の構造規則にについての議論をおさらいしましょう。Reflexivity の規則はありえない状況の最小単位を提示する構造規則で、それが FDE においては (Reflexivity P) と (Reflexivity N) の二つになるのでした。他の二つの規則、(Re-

flexivity L) と (Reflexivity R) は FDE の直観に沿わないとして構造規則に加えませんでしたが、実はこれらを構造規則として認めると古典論理になります。「真である」「偽でない」と「真でない」「偽である」がそれぞれ潰れてしまうんですね。例えば以下の証明は $a, A : b \vdash c : d$ から $a : b \vdash c : d, A$ を導いていて、命題 A を「真である」から「偽でない」へ移しています。

$$\dfrac{\dfrac{\dfrac{}{: \vdash A : A}\,\text{(Reflexivity R)}}{a : b \vdash c, A : d, A}\,\text{(Weakening)} \quad \dfrac{a, A : b \vdash c : d}{a, A : b \vdash c : d, A}\,\text{(Weakening)}}{a : b \vdash c : d, A}\,\text{(Cut P)}$$

証明に (Reflexivity R) を使っているのがミソです。このように古典論理の双シーケント計算を考えても 4 つのパートが実質 2 つに潰れてしまうことになるのであまりメリットはないように思えます。

ちょっと待ってください、この議論を別の視点からとらえなおしてみます。通常の古典論理のシーケント計算では以下の構造規則が Reflexivity に関する唯一の構造規則です。

$$\dfrac{}{A \vdash A}\,\text{(Reflexivity)}$$

4.1 節にあるシーケントの気持ちを思いだしてほしいのですが、このシーケントは「A が真でかつ偽であることはありえない」ということを表わしています。さらに古典論理では矛盾は「A が真でかつ偽」の形しかないので、(Reflexivity) は矛盾が存在しいないことを表す構造規則だと解釈できます。しかしシーケントから双シーケントに拡張すると、矛盾をより細かい粒度で表現することが可能になります。

- $\bot_P$: A が真でかつ真でない
- $\bot_N$: A が偽でかつ偽でない
- $\bot_L$: A が真でかつ偽である
- $\bot_R$: A が真でも偽でもない

双シーケント計算における Reflexivity の P,N,L,R の各規則はそれぞれ以下の矛盾を認めない規則であると解釈できます。古典論理ではこれらを全て認めていないのに対して、FDE では $\bot_L$ と $\bot_R$ を**真なる矛盾**として認めていて、これは (Reflexivity R) と (Reflexivity L) の構造規則を落とすことによって達成されているわけです。

まとめると、FDE の本質的な部分は

- 双シーケントによって古典論理における矛盾 (=Reflexivity の規則) を 4 つに細分化し
- そのうち 2 つを**真なる矛盾**として認める (= 構造規則に入れない)

部分にあるのではないかと思われます。このような見方をすると FDE は古典論理の「部分構造論理」の一種として考えられないでしょうか。

5. まとめ的なものと読み物

お疲れさまでした。ここまで読んでくれてありがとう、ありがとう .....

真矛盾主義という名前のイカツさとか主張の過激な響きとは裏腹に、実は結構ちゃんとした動機づけや形式的体系があるという雰囲気を感じとってくれたら幸いです。こういうギャップがいいんですよ、わかっていただけますよね。

この記事では真矛盾主義のエッセンスの部分を解説しましたが、説明できていない部分 (あるいはまだ寄稿者が理解できていない部分) がかなりあります。 真矛盾主義についてもっと詳しく知りたいという方のために文献の紹介をします。

[1] は論理学の素養がなくても読める入門書です。 前に話したように論理学の「なぜこういう論理を考えたいか」というところをわかりやすく説明してくれているので論理をかじったことのある人が読んでも面白いと思います。 この本で嘘つきのパラドックスについて論じている章があって、その中で真矛盾主義について言及されています。ちなみに入門書なのに真矛盾主義が入ってきているのは著者の Graham Priest が真矛盾主義の研究者だからだと思われます。

[2] は同じく Graham Priest による非古典論理一般についてのぶ厚い教科書で、本当にぶ厚いので持ち運ぶ気が失せます。結構な数の非古典論理を紹介していていて、その一つとして FDE も紹介されています。[1] は気持ちの説明だけですが、こちらの本ではいくつかの意味論とタブロー法による推論、 他の多値論理との関わりについてちゃんとした形式的な定義とともに書いてあります。

[3] は FDE の初出の論文です。1977 年の論文で動機として AI が出てきているのは時代

背景を反映している気がします[*26]。今回の話に関係ありそうだった最初のパートしか読んでません。このパートでは FDE の意味論を主に解説しており、最後に少しシーケント計算風の推論体系について言及しています。ちなみにその推論体系は今回紹介したものと違って二つのパートからなるシーケントを用いていますが、[4] でそれらの関連性が論じられています。

[4] は Biconsequence Relations という枠組を FDE や古典論理やその他の多値論理を論ずるための共通の土台として使おうという主旨の論文です。Biconsequence Relation の枠組は結構よくできていて、FDE の推論体系の体系化としてほぼ完成しているのではないかという気さえします。FDE の推論体系に興味があるのであれば必読の論文と思われます。ちなみに今回の記事は FDE の紹介をするのが主旨でしたので FDE に関する部分だけを切り出してシーケント計算の体系にした上で紹介しています。

参考文献

[1] Graham Priest. *Logic: A Very Short Introduction*. Oxford University Press, 2017.

[2] Graham Priest. *An Introduction to Non-Classical Logic: From If to Is*. Cambridge University Press, 2008.

[3] Nuel Belnap. A Useful Four-Valued Logic. *Modern Uses of Multiple-Valued Logic*, 2, pages 5–37, 1977.

[4] Alexander Bochman. Biconsequence Relations: A Four-Valued Formalism of Reasoning with Inconsistency and Incompleteness. *Notre Dame Journal of Formal Logic*, 39(1), pages 47–73, 1998.

26 攻殻機動隊 SAC の中で、タチコマがアンドロイドに嘘つきのパラドックスを言うシーンがあるんですよ。アンドロイドは嘘つきのパラドックスが真か偽か判定しようとして無限ループに陥ってフリーズして、それをタチコマが面白がるわけです。あのアンドロイドの思考エンジンは FDE を実装してなかったんやろうな .....

あとがき

この度は yabaitech.tokyo vol.3 をお買い上げいただきありがとうございます。yabaitech.tokyo は大学院時代の同期によって結成されたサークルで、コンピュータに多少なりとも関係する話題について好きなことを記事にして寄稿し合同誌を作るというゆるいコンセプトのものです。3 つ目の合同誌となる本誌には群論、オートマトン、OS、粘菌、論理と様々な分野の記事が集まりました。忙しい中時間を確保して記事を書いてくれた寄稿者の各位をはじめ、この本を出版し技術書典に出展するにあたって協力してくれた yabaitech.tokyo の皆に感謝します。

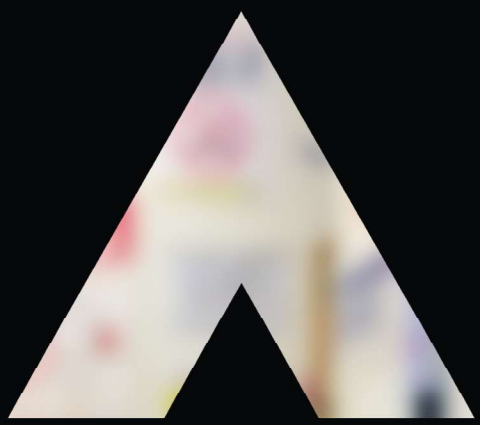
目次を見てもわかるように、それぞれの記事は寄稿者の興味を反映し尖ったものになりがちです。書きたいことを書いていたらそうなったという話ではあるのですが、こんなニッチなことを書いていて果たして読者の関心を得ることができるだろうかというのは寄稿者の心配の種だったりします。この点については vol.2 の反省会で話し合って「尖った部分はそのままに、でも読みたい人がちゃんと読めるように」を目指そうという結論になりました。そういう試みの一環として今回は脱稿前に 2 週間のピアレビュー期間を設けてある程度の記事の品質の担保を図っています。yabaitech.tokyo vol.3 が目標を達成できたかはまだなんともいえませんが、あなたの興味を惹き好奇心を満たす記事があれば幸いです。

サークルカットと表紙について　サークルカットは gfn によるもので、知ってる人は知っている gfn ガールズのイラストです。表紙のデザインは censored によるものです。もとは yabaitech.tokyo のメンバー (最近加入しました) が写っていた写真なのですが、photoshop の力によって陽菜ちゃんに置き換えられました。

(文責 : zeptometer)

yabaitech.tokyo vol.3

発行日	2019.09.22
サークル	yabaitech.tokyo
Web サイト	http://yabaitech.tokyo
連絡先	admin@yabaitech.tokyo
印刷所	有限会社 ねこのしっぽ



Schreier-Sims のアルゴリズムを Rust で実装した
koba-e964

はじめての AI —AI は AI でも Automata Inference の方だがなぁぁーっ !—
MasWag

Writing a (micro)kernel in Rust in 12 days - 2.5th day -
nullpo-head

粘菌で計算がしたい!
wasabiz

真矛盾主義入門
zeptometer

---

yabaitech.tokyo vol.3