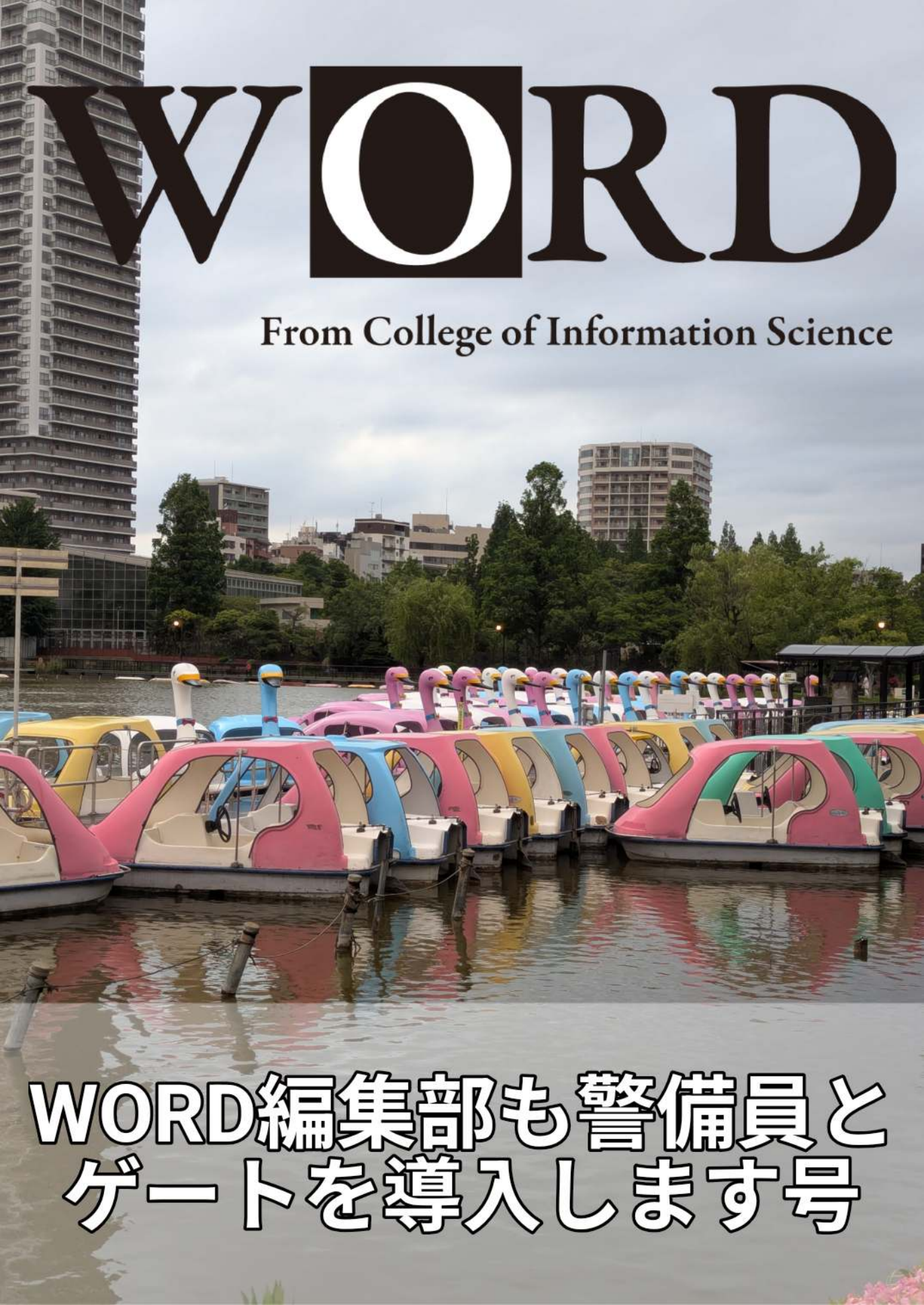
WORD

From College of Information Science



WORD編集部も警備員とゲートを導入します号

目次

LaTeX のアレを Typst でどうやるか

文　編集部　Azumabashi

1　さよならLaTeX テンプレート

WORD 57 号（今号）より，従来の WORD 向けLaTeX テンプレートが廃止され，WORD は Typst で組版されることとなった．ということは，Typst の使い方を学習する必要がある！そこで，この記事では，「LaTeX で実現できる○○は Typst ではどう実現できるのか？」を，特に論文やレポート執筆で使いそうな範囲に絞ってまとめてみようと思う[†1].

もっとも，ここで問題になるのが，**Typst にはいまだ決定版と言えるテンプレートが存在しない**ことである．素の（一切設定を加えていない）ソースファイルをコンパイルしても，日本語組版に適した設定にはなっていないので，日本語の文書としては少しおかしな結果が得られる．日本語組版を前提とした設定は，日本語組版向けスターターパックのようなパッケージ（LuaLaTeX の `luatexja` パッケージが代表例）で提供されるべきものだろうが，そういう汎用パッケージはまだ提供されていないと思われる．一応 `jarticle` 風のテンプレートは存在する[†2] し，『美文書』の奥村先生によるテンプレートもある[†3] ものの，`jarticle` のような「決定版」の地位を占めている訳ではないようだ．本稿では，WORD のテンプレートで提供された環境上で色々試してみることとする．

2　インストール

Typst を使うためには，処理系のインストールが必要である．厳密にいうと，LaTeX での Overleaf のようなオンラインエディタが公式で提供されているため，ただ触るだけなら処理系のインストールは不要である．しかし，手慣れたエディタを手放したくないので，今回は Typst をローカルにインストールする．インストール方法は Typst のリポジトリ[†4] にまとめられており，例えば macOS なら素直に

```
$ brew install typst
```

とすればコマンド `typst` が利用可能になる．本稿執筆時点の最新版はバージョン 0.13.1 である．

ついでにエディタの設定をしておこう． Emacs の場合は，typst-ts-mode[†5] というパッケージを導入することで，シンタックスハイライトが有効化される． MELPA には登録されていないので，導入はちょっとだけ厄介であるが，leaf を使っている場合は次でできる：

```
(leaf typst-ts-mode
```

†1　実のところ https://typst.app/docs/guides/guide-for-latex-users/の二番煎じと言えばそうなのだが…….
†2　https://github.com/satshi/typst-jp-template
†3　https://typst.app/universe/package/js
†4　https://github.com/typst/typst
†5　https://git.sr.ht/~meow_king/typst-ts-mode

```
  :el-get (typst-ts-mode :type git :url "https://git.sr.ht/~meow_king/typst-ts-mode")
  :mode "\\.typ"
  :custom
  (typst-ts-mode-enable-raw-blocks-highlight . t))
```

`:el-get` でどのリポジトリから持ってくるかを指定していること以外は普通の設定であろう．そして，

```
(typst-ts-mc-install-grammar)
```

を`*scratch*`かどこかで評価して Typst の文法を導入する．

VS Code や Vim にも何かはあるだろうが，調べていない．なにしろ普段使いが Emacs なので……[†6]．

コンパイルは，

```
$ typst compile file.typ
```

でできる． `compile` というサブコマンドがあることに注意． LaTeX で言うところの `dvipdfmx` や `bibtex` が行っていたことも，すべてこのコマンドがやってくれる．

3 LaTeX のアレって Typst でどうやる？

それでは，LaTeX でよく見る「アレ」を Typst ではどうやるのか見てみよう．

3.1 節構造

見出しは`=`を連ねる形になる．すなわち，`=`が最大の見出し（LaTeX での`\section`），`==`がその次に大きな見出し(`\subsection`)，といった具合である． LaTeX とは逆に，デフォルトでは番号がつかない．番号をつけたければ，次のようにする：

```
#set heading(numbering: "1.a.1.")
= section
== subsection
=== subsubsection
```

`numbering` で，どういう書式で番号を付与するかを指定する．受け付けられる書式は，公式ドキュメント[†7] にまとめられているので，詳細はそちらに譲る．組版結果は省略するが，本稿の組版結果を見ればだいたいわかるだろう．

なお，便宜上`=`が`\section` で云々と書いたが，本当は見出し間の大小の順序関係のみが存在するようである．したがって，適当にカスタマイズすれば，`=`で`\chapter` 相当の出力をさせることも可能である．もっとも，学位論文でもなければ`=`は`\section` に対応すると考えてよいだろう．

†6 SKK と `yatex` からはもう逃れられない．

†7 https://typst.app/docs/reference/model/numbering/

3.2 テキストの装飾

Typst でも，一部のテキストを**太字**にしたり，*italic* にしたり，`monospace` なフォントを使ったりできる．このためのマークアップは非常に単純である：

```
Typstでも，一部のテキストを*太字*にしたり，_italic_にしたり，`monospace`なフォントを使ったりできる．
```

かなり Markdown ライクに書ける．ちなみに，改行に関する規則は LaTeX と同様のようである．すなわち，

```
ここは
一文に
なります．
もうひとつ．

改段落されました．
```

というテキストは，

ここは一文になります．もうひとつ．

改段落されました．

となる．

テキストに色をつける場合（`\textcolor` に相当）は，次のようにする．

```
テキストに
#text(fill: red)[色]
をつける場合は，次のようにする．
```

しばらくの間色を変えたいとき（`\color` に相当）は，次のようにする．

```
ふつうの文です．

#set text(fill: blue)
ここはずっと青色になります．
ここも青色．

#set text(fill: black)
ここからまた黒に戻ります．
```

3.3 数式

LaTeX の優れている点のひとつが数式の表現力の高さであることには異論はないだろう．Typst でも綺麗な数式を組版することができる．例えば，「関数$f : \mathbb{R} \to \mathbb{R}$が区間$[0, 1]$で連続であるとき，

$$\int_0^1 f(x)dx = \lim_{n\to\infty} \sum_{k=0}^{n} f\left(\frac{k}{n}\right) \tag{1}$$

である．」という文章は，次のように書く．

```
関数$f: RR -> RR$が区間$[0,1]$で連続であるとき，

```

```
$
    integral_0^1 f(x) d x = limits(lim)_(n -> infinity) sum_(k=0)^n f(k/n)
$

である.
```

数式の表現方法はLaTeX と全く異なると言ってもよいだろう． LaTeX におけるインライン数式もディスプレイ数式も，`$` で数式を挟むことで実現されるが，`$` の直後にスペースが**ない**とインライン数式となり，スペースが**ある**とディスプレイ数式となる．例えば，「関数`$ f $`」と書いてしまうと，fはディスプレイ数式として扱われる．また，複数の文字の積（に見える項）では，文字をひとつずつ離して書く必要がある．例えば，aとbとcの積abcは，`$a b c$` であって `$abc$` ではない．

上付き文字や下付き文字の構文はLaTeX と同様だが，どこまでが上付き（下付き）になるかが異なる．例えば，「aの15乗とbの20乗の和」$a^{15} + b^{20}$は，普通に `$a^15 + b^20$` と書けば，15がaの指数であると判断してくれる．もし「aの$15 + b^{20}$乗」$a^{15+b^{20}}$を組版したければ，指数部分を丸カッコで囲って，`$a^(15+b^20)$` とする．だから，

$$(f + g_{a+b})^{-1}_{c,d}(x) = e^{x(1+n)} \tag{2}$$

という数式は，`(f + g_(a+b))^(-1)_(c,d)(x)=e^(x+(1+n))`と書けばよい．右辺のように，丸カッコが複数の意味で使われていても，いい感じに解釈してくれる．

工学系で見られる「ディーが立体の$\mathrm{d}x$」は，`$dif x$` とする．黒板太字による集合（例えば$\mathbb{R}$や$\mathbb{N}$といったもの）は，`$RR$` のように文字を 2 回書くとできる．写像の定義域と値域を明示する記法$f : U \to V$におけるコロンは，LaTeX ではコロン`:`ではなく，コマンド`\colon` を使うのが正式である（ただの`:`は$2 : 3$のような比を表すコロンである）． Typst でも `colon` があるが，`$f colon U -> V$` と書いても$f : U \to V$となってしまい，`\colon` のような振舞いをしない． Reddit で見かけた解決法として，

```
#show sym.colon: $class("fence", colon)$
```

と書いておく[†8]，というものがあり，この設定のもとでは $f\colon U \to V$と正しく組版される．ただし，その場合でも`:`を使う必要があり，`colon` を使ってしまうと$f : U \to V$と結果が変わらないことになる．

別の例を示してみよう．

$$\forall \varepsilon > 0, \exists \delta, |x - a| \leq \varepsilon \Longrightarrow |f(x) - A| \leq \delta \tag{3}$$

は，次のように書く．

```
$
    forall epsilon>0, exists delta,
    |x-a| <= epsilon ==> |f(x)-A| <= delta
$
```

†8 https://www.reddit.com/r/typst/comments/17asyg4/how_to_typeset_a_colon_in_math_mode_correctly/

先程は触れなかったが，<= や ==> など，実際に出力される記号に近しい記号を用いることができる． ==>は長い矢印（\implies に相当）を表現するためであり，短いもの⇒であれば $=>$ で足りる．不等号系の記号と矢印系の記号では=と>の順序が逆である．つまり，$=>$ は⇒であり，$<=$ は≤である．したがって，左向きの矢印（←や⟸）にはこの記法は適用できない．このような矢印のためには，$arrow.l$ や $arrow.l.double.long$ とする．順序は重要ではないようで，$arrow.l.long.double$ でも⟸となる．

ギリシャ文字は，数学で典型的な読み方を直接書けばよい．つまり，$\alpha\beta\gamma$は $alpha beta gamma$ である（半角スペースに注意）．イプシロンの記号は，標準のεは\varepsilon 相当の記号を出力する． LaTeX での\epsilon のためには，epsilon.alt を用いればϵとできる．ちなみに，$ε$ と書いてもεになるが，通常の（JIS 配列または US 配列の）キーボードで Unicode 文字を入力する手間を考えると，メリットは微妙かもしれない．

このほかの数式もいくつか見てみよう．

$$
\begin{aligned}
(0,1) =]0,1[= \{a \in \mathbb{R} \mid 0 < a < 1\} \\
(0,1) \subseteq (0,2) \subseteq \mathbb{C} \\
\{C \in \mathcal{W}^t; |C| \bmod 2 = 1\} \\
\delta(i,j) \coloneqq \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise} \end{cases}
\end{aligned}
\tag{4}
$$

これは，次のように書けば得られる．

```
$
    (0,1) = ]0,1[ &= {a in RR | 0<a<1}\
    (0,1) &subset.eq (0,2) subset.eq CC \
    { C in cal(W)^t &; |C| mod 2 = 1 } \
    delta(i,j) &:= cases(1 "if" i=j, 0 "otherwise")
$
```

ディスプレイ数式で位置揃えをするためには，& 記号を使う．改行は\でできる． $\mathcal{W}$のようなカリグラフィックなフォントで文字を出力するためには，cal を使う． cal(W)^t がある特定の構造（例えば，ある時刻tにおけるグラフの弱連結成分(weakly conncted components)の集合）を表し，これが文章中に繰り返し現れる場合は，いちいち cal(W)^t と書かないで，\wccs のようなコマンドが使えると便利である．これは，次のようにすればよい．

```
#let wccs = $cal(W)^t$

$forall C in #wccs, |C| >= 1$
```

そうすれば，「$\forall C \in \mathcal{W}^t, |C| \geq 1$」とできる．もし上付き文字の$t$を引数で指定したい場合は，

```
#let wccs2(t) = $cal(W)^(#t)$
```

とすれば，$#wccs2($T$)$ で$\mathcal{W}^T$とできる．角カッコを使って #wccs2[T]と書いても$\mathcal{W}^T$と同じ出力になる．引数を明示的に $ で囲まないといけないようで，これを忘れて #wccs2(T)と

書くと T が変数扱いとなってしまうし，`#wccs2[T]`と書くと$\mathcal{W}^{\mathrm{T}}$と T が斜体にならない．例えば，時刻t^2を渡してみると違いが顕著である：$\mathcal{W}^{t^2}$と$\mathcal{W}^{t\hat{}2}$.

LaTeX の気分で書くと，`cal(W)^t` を `$` で挟むのはどこか気持ち悪いのだが，こうしないとコンパイルエラーになる．事実上 `$` がネストしている形になるのだが，それはよいのだろうか…….

場合分けは `cases` 関数を使う．条件部分は，`if` などの文字列を置いた後に書く．どうやら記号類ではない文字列を置くのは必須のようで，これがないと，x^2が正のときは1，そうでなければ0を返す関数$f(x)$の定義が

$$f(x) = \begin{cases} 1 x^2 > 0 \\ 0 x^2 \leq 0 \end{cases} \tag{5}$$

となり，詰まってしまう．高校までの数学でおなじみの「条件部分をカッコで括る」表記法を試してみると，これも

$$f(x) = \begin{cases} 1 (x^2 > 0) \\ 0 (x^2 \leq 0) \end{cases} \tag{6}$$

のように詰まってしまうらしい．

なお，集合を組版してみると必ず試したくなるのが，半開区間$\left\{a \in \mathbb{R} \mid 0 < a < \frac{1}{2}\right\}$である．

$$\left(0, \frac{1}{2}\right) = \left\{a \in \mathbb{R}; 0 < a < \frac{1}{2}\right\} = \left\{a \in \mathbb{R} \mid 0 < a < \frac{1}{2}\right\}. \tag{7}$$

多くのカッコは，そのカッコの中身の大きさに従って自動伸縮する（`\left` と`\right` がデフォルトで効いている状態である）．ただし，LaTeX と同様に，単に縦棒を置いただけでは，その縦棒は伸びてくれない． LaTeX ではとりあえずの対処として，`\mid` の代わりに `\mathrel{}\middle|\mathrel{}`を使うという方法が知られているが，Typst で同様のことを綺麗に解決する方法はどうやらないらしい．

また，集合の半開区間の記法には(a, b)のほかに$]a, b[$もあるのだが，これを Typst ではうまく扱えないらしい． 式 4 の組版結果を見ると，イコールと閉じカッコ]の間隔が狭すぎる．もっとも，デフォルトのLaTeX でもうまく扱えない（ちゃんとやるためには `interval`パッケージが必要）のだが…….

最後に，ベクトルと行列を見ておこう．

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \alpha \mathbf{1} \tag{8}$$

は，次のように書く．

```
$
    mat(a_1, a_2; a_3, a_4) vec(x,y) &= alpha bold(1)
$
```

行列は mat，ベクトルは vec で書く．行列の場合，同じ行にある要素は,で区切り，改行は;で区切る．ベクトルの場合はすべて,で区切る．行列もベクトルも似たようなものと思いがちだが，Typst の場合微妙に組版結果が異なる．以下の数式の 1 行目は mat，2 行目は vec で組んでいるが，違いは一目瞭然だろう．

$$\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} x + \alpha \\ y + \beta \end{pmatrix}$$
$$\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} x + \alpha \\ y + \beta \end{pmatrix} \tag{9}$$

縦ベクトルを行内で書く場合，転置記号を使って$(1,2,3)^\top (\in \mathbb{R}^3)$とすることもしばしばあるが，横ベクトルを書くための記法はどうやらないようで，`$(1,2,3)^top$` とするのが一番よいだろう．無理やり行列を使って書くと，$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}^\top$となって，潰れてしまう．ベクトルや行列を角カッコにしたい場合は，

```
#set math.mat(delim: "[")
```

のように宣言すれば

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{10}$$

となる．

太字のベクトルは bold を用いて表現できる．矢印によるベクトルは，特別な関数が用意されていないようで，`arrow(v)`と書いて$\vec{v}$とするしかない．

ちなみに，数式に振った番号の参照もできるが，デフォルトでは数式に番号がつかないので，参照できないということになっている（無理にやろうとするとコンパイルエラーになる）．この記事では，

```
#set math.equation(numbering: "(1)")
```

と設定してあるので，数式番号が横についている．このような設定があれば，数式

$$A\boldsymbol{x} = \lambda \boldsymbol{x} \tag{11}$$

を，式 11 として参照できる．これは，次のように書けば得られる．

```
このような設定があれば，数式

$
    A bold(x) = lambda bold(x)
$ <math>

を，@math として参照できる．
```

foo というラベルの付与（LaTeX での`\label{foo}`）は`<foo>`として数式の直後に置き，このラベルの参照（LaTeX での`\ref{foo}`）は`@foo` とする．「式」は自動で補ってくれる． 式 4 のようにいくつかの数式が同一の `$ ... $` に含まれる場合の挙動はLaTeX と少し異なる． 1 つのブロックに，数式番号は 1 つしかつかないらしい．したがって，式 4 の上から 2 番目の式

$((0,1) \subseteq (0,2) \subseteq \mathbb{C})$ だけをピンポイントで参照する方法は，おそらくない．これだけを参照したい場合は，この式を独立したディスプレイ数式として組版する必要がある．

なお，数式番号をつけたくない場合，例えば

$$a^3 + b^3 \leq 100$$

のような数式を混ぜるためには，次のようにすればよい．

```
#math.equation(block: true, numbering: none)[
    $ a^3 + b^3 lt.eq 100 $
]
```

`#math.equation[ ... ]`の中ではダイレクトに数式が書けそうな気もするが，書けないようである．なお，LaTeX での`\nonumber` における振舞いをする仕組みはないようである．

3.4 グラフィックス

論文などでよく図を挿入したいことがある．いくつかの方針が考えられるが，

- 外部のツールで画像を作成し，Typst に読ませる
- Typst にグラフィックスを作成させる

の 2 通りある．両方見てみよう．

既存の図を読ませる方針

ここでは，誰でも知っている$y = x^2$のグラフを Gnuplot で雑にプロットさせたものを読み込ませてみる．例えば図 1 のような調子である（可視化結果としてはまったく十分ではないが，本稿の主題はうまい可視化結果を提供することにはないので，ご容赦を）．

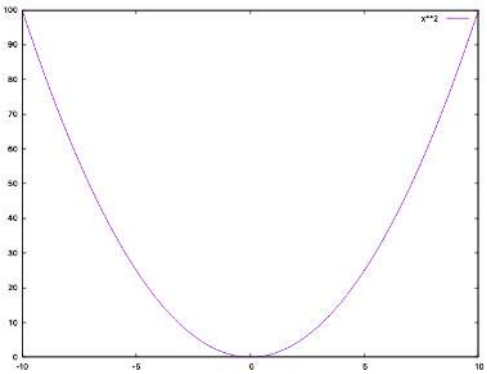


図 1: $y = x^2$のグラフ．

これは，次のようにすれば得られる．

```
例えば@label のような調子である．

#figure(
```

```
4     image("fig/x2.png", width: 50%),
5     caption: [$y=x^2$のグラフ．]
6 ) <label>
```

`image` 関数の第 1 引数に画像ファイルへのパスを渡す． `width` は`\textwidth` に対する割合を指定する．次の図のように，幅を絶対指定することもできる．

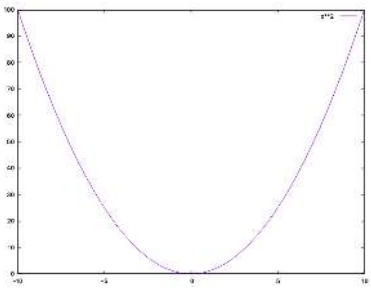


図 2: `width` を `5cm` と絶対指定した図.

「図 1」の「図」は，ラベルが参照するときに自動でつけてくれる．ラベルの付け方とその参照方法は，数式と同様である．

`#figure` では，標準的な画像形式（JPG と PNG）を受け付けてくれる．しかし，多くの場合，論文で取り込むのはラスタ形式の図ではない． PDF に代表される，ベクタ形式の図を取り込みたいことのほうが多いだろう[†9]．重大な注意点として，**Typst では PDF を画像として読み込むことができない**ことがある[†10]．ベクタ形式で読み込みたいなら，SVG にする必要がある． 図 3 のように SVG 形式を読み込む場合も，PNG を読み込む場合と同様にできる．電子版で WORD を読んでいる読者は，PDF をうんと拡大してから図 1 と図 3 を比べてみてほしい． 図 3 では画像の「ジャギり」がないはずである．

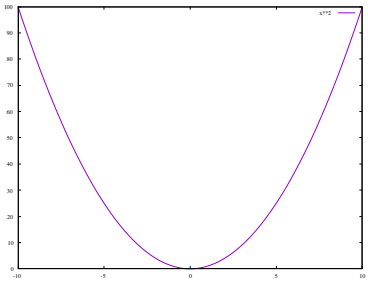


図 3: SVG 形式の図を読み込ませる例.

†9 PDF で拡大して読むときに，そちらのほうが綺麗になるので.
†10 https://github.com/typst/typst/issues/145

なお，デフォルトでは，図は `#figure` が書かれた位置に置かれる（LaTeX の `H` と同じ挙動）．これをページの上端または下端に置かせるためには，`placement` に `top` または `bottom` を渡す．自動で判別させるためには，`auto` を渡す．例えば，図 4 は，ソースファイル上ではこの文の直下に書かれているが，実際に置かれているのはこのページの上である……はずなのだが，なぜか `#figure` にある位置（この段落の直後）に置かれている．どうしてなんだろう．

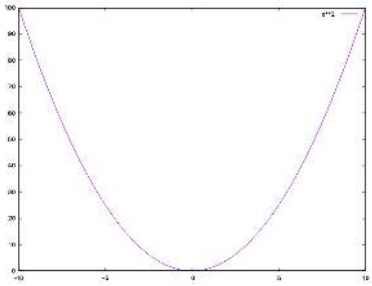


図 4: `placement` に `top` を指定した図の例．

Typst に図版ごと組ませる方針

LaTeX には TikZ という狂った[†11] パッケージが存在することは有名である．このパッケージの力を借りれば，かなりの図版を LaTeX のソースファイルだけで実現できる．

Typst で TikZ に相当するパッケージには，CeTZ[†12] がある． CetZ ではなくて CeTZ のようで，TikZ の気分で書いていると T を小文字にしてしまう．では，CeTZ で試しに図版を作ってみよう（図 5）．

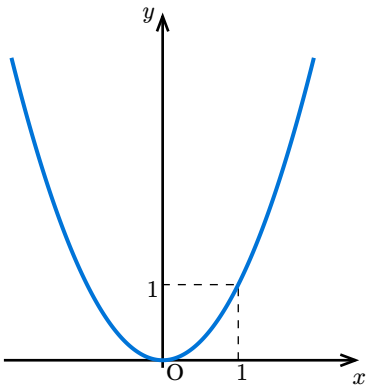


図 5: CeTZ で描画した，$y = x^2$のグラフ．

†11 褒め言葉です．

†12 https://typst.app/universe/package/cetz/

できた．以上のプログラムは，次のようにすれば得られる．

```
#import "@preview/cetz:0.4.0"
#import "@preview/cetz-plot:0.1.2"

#let f(x) = { x * x }

#figure(
    [
    #cetz.canvas({
        import cetz.draw: *

        line((3,0), (3,1), (2,1), stroke: (dash: "dashed", thickness: 0.5pt))
        content((2.05,-0.05), "O", anchor: "north-west")
        content((3.05,-0.05), [$1$], anchor: "north")
        content((1.95,0.95), [$1$], anchor: "east")
        cetz-plot.plot.plot(
            size: (4, 4),
            x-label: $x$,
            y-label: $y$,
            x-min: -2,
            x-max: 2,
            y-max: 4,
            x-tick-step: none,
            y-tick-step: none,
            axis-style: "school-book",
            cetz-plot.plot.add(
                style: (stroke: blue + 1.5pt),
                domain: (-2, 2),
                samples: 200,
                f
            )
        )
        })
   ],
    caption: [CeTZで描画した，$y=x^2$のグラフ．]
)
```

CeTZ は Typst の標準ライブラリではないので，`#import` により読み込む．標準ライブラリでないパッケージは，Typst Universe[†13] から検索できる． `:`の後ろにあるのはバージョンであり，これが明記してあることで再現性が保たれる．

CeTZ を使って図版を作るためには，基本的には Getting Started のページ[†14] にある雛形に従えばよい． CeTZ で任意の関数を描画するためには，当該の関数を `#let` で定義してから使うのが便利だろう．もちろん，`x => x * x` のような無名関数スタイルでもよい．関数を定義する場合は，最後に評価された式が戻り値となるようである．

直線は `line` で描画する．点の座標の列を渡すと，その点を順番に結ぶ直線が描画される．指定された座標がおかしいように思えるが，$(0,0)$とは最も左下を表すようであり，図版の中で点$(1,0)$と$(1,1)$を結ぶ線分を描きたくても，Typst に渡す座標はまた別のようである．破線

†13 https://typst.app/universe/

†14 https://cetz-package.github.io/docs/getting-started

にしたい場合や太さを調整したい場合は，`stroke` 以下で調整する． TikZ であったような `thick` や `ultra thick` のようなものはないらしい．アンカー（指定された座標に対して文字を東西南北どこに配置するか）を指定することができるが，なぜか東西南北が逆になっている．文字を置きたい場合は `content` を使う．

論文でよくあるのは，図として掲載して，後で「図 5」のように参照するというものだろう．これは単純に，CeTZ のコードを`[ ... ]`の中に置いて，`image` 関数の代わりに `#figure` に渡せばよい（ソースコード例を参照）．

TikZ と同様に，サンプルとなる図表とそのソースコードを公開しているページがある[†15]．場合によっては，TikZ と CeTZ の両方のコードが入手できる場合もある．……が，ひょっとして TikZ のほうが簡単なのでは？

3.5 表

論文でよくあるもののひとつが表である．例えばこんな感じのもの：

A	B
ほげ	ふが
ぴよ	改行も かんたん

これは次のようにすれば得られる．

```
#table(
    columns: 2,
    table.header([A], [B]),
    align: left,
    [ほげ],
    [ふが],
    [ぴよ],
    [改行も\ かんたん]
)
```

列数は `columns` でセットする．文字列の配置は `align` で指定する．それぞれのセルの中身は`[ ... ]`に入れる．改行は`\`を使えば簡単にできる．

デフォルトの表では，すべてのセルが線で囲まれる．これを，例えばLaTeX の `booktabs` パッケージのように一部だけに線を引いた体裁にするためには，ちょっとしたプログラミングが必要である．列によって文字の揃え方を変更したい場合（例えば，一番左だけは左揃えで，残りは中央揃えしたいとき）も同様．ちょうど公式ドキュメントに似た例があるので[†16]，詳細はそちらに譲ろう．

ちなみに，Typst の `#table` はLaTeX の `tabular` にしか相当しない．つまり，これだけではキャプションがつけられないし，参照もできない（なんてこった）！こういったことをするた

†15 例えば https://diagrams.janosh.dev/など．
†16 https://typst.app/docs/reference/model/table/

めには，`#figure` を流用すれば，表 1 のようにできる．表を表示するために `#figure` を使う，というのはやや微妙さを感じないでもない[†17]．

A	B
ほげ	ふが
ぴよ	改行も かんたん

表 1: `#figure` を流用して参照できるようになった表.

自動でキャプションが「表」になるのは良い点だが，キャプションが表の下につくスタイルはいまだ標準とは言い難い．キャプションを上に配置するためには，

```
#show figure.where(kind: table): set figure.caption(position: top)
```

を `#figure` の前に置く．実際にこれを実行すると，表 2 が得られる．

表 2: キャプションを表の上に配置するよう設定を変更した表.

A	B
ほげ	ふが
ぴよ	改行も かんたん

まとめると，次のように書けばよいということになる．

```
// キャプションを表の上に置くための設定（1回だけ行えばよい）
#show figure.where(kind: table): set figure.caption(position: top)

// キャプションやラベルが設置できる表
#figure(
    caption: [`#figure`を流用して参照できるようになった表．],
    [
        #table(
    columns: 2,
    table.header([A], [B]),
    align: left,
    [ほげ],
    [ふが],
    [ぴよ],
    [改行も\ かんたん]
)
    ]
) <table>
```

†17 LaTeX でも表も図も float という同じ概念に従って管理されるが，`figure` 環境と `table` 環境は別物である．

3.6 tcolorbox 的なボックス

LaTeX では，一昔前までは ascmac パッケージで提供されていたようなシンプルなボックスが広く使われていたが，最近では tcolorbox によるカラフルなボックスもよく使われるようになってきた． Typst における似たようなボックスは，colorful-boxes[†18] というパッケージで提供されている．似たことは showybox[†19] でもできる．

タイトル

ちょっと豪華なボックス．

これは，次のようにすれば得られる．

```
#import "@preview/colorful-boxes:1.4.3": colorbox

#colorbox(
  title: [タイトル],
)[
  ちょっと豪華なボックス．
]
```

import 文の後ろに「: colorbox」がついているが，これがあると，Python の from package import func と同じような機能を果たす．例によってこのボックスも色々とカスタマイズができるが，本稿では一旦ここまでに留めておく．

3.7 定理環境

研究内容によっては，いわゆる「定理環境」が必要になることがある．つまり，

Definition 1.1 グラフ

グラフGとは，2 つ組(V, E)のことである．ここで，Vは頂点集合，$E \subseteq V \times V$は辺の集合を表し，$(u, v) \in E$とは頂点$u \in V$から頂点$v \in V$への辺が存在することを意味する．

Definition 1.2 無向グラフ

無向グラフとは，グラフ$G = (V, E)$であって，

$$\forall (u, v) \in E, (v, u) \in E \tag{12}$$

を満たすものである．

†18 https://typst.app/universe/package/colorful-boxes/
†19 https://typst.app/universe/package/showybox

Theorem 1.3 **すごい定理**

任意の無向グラフ$G = (V, E)$について，$|E|$は偶数である．

Proof: 明らか． □

のようなものである．このためには，`thmbox` パッケージ[†20] が必要である．使い方はほぼ同じなので，Definition 1.1 の使い方だけを以下に示す：

```
#import "@preview/thmbox:0.2.0": *

#show: thmbox-init()

#definition(title: [グラフ])[
    グラフ$G$とは，2つ組$(V,E)$のことである．
    ここで，$V$は頂点集合，$E subset.eq V times V$は辺の集合を表し，$(u,v) in E$とは頂点$u in V$から頂点$v in V$への辺が存在することを意味する．
] <graph>
```

`#show: thmbox-init()`はいわゆる「おまじない」．デザインなどはデフォルトのままであるが，かなりいい感じである．見出しは，「定義」ではなく英語の「Definition」になっている．日本語の「定義」を使いたい場合は，`ctheorems` パッケージ[†21] を使ったほうがいいかもしれない．このパッケージを使う場合の例を定理 1.3.7.1 に示す．

定理 1.3.7.1 (タイトル): `ctheorems` による定理．

証明: 証明． ■

これは次で得られる．

```
#import "@preview/ctheorems:1.1.3": *

#let ctheorem = thmbox("定理", "定理")
#ctheorem([タイトル])[
    `ctheorems`による定理．
] <ctheorems>
#let cproof = thmproof("証明", "証明")
#cproof[
    `ctheorems`による証明．
]
```

`theorem` ではなく `ctheorem` を用いているのは，`thmbox` による `#theorem` との衝突を避けるためである． `cproof` のほうも同様．こちらのほうが，よりLaTeX の `amsthm` に近い挙動・見た目

†20 https://typst.app/universe/package/thmbox/
†21 https://typst.app/universe/package/ctheorems/

だろう．ただし，定理 1.3.7.1 のように，無駄に広々とした空白が付与されるので，これを削るためにはカスタマイズが必要．

3.8 文献の引用

論文を書くときには，当然文献を引用することになる．例えばこんな感じで：[1] [2]．このためには，LaTeX で使っていた BibTeX ファイルをほぼそのまま流用できる．「ほぼそのまま」と書いたのは，LaTeX 向けの記述（例えば`\LaTeX` など）は修正する必要があるからである．実際，本記事で用いている bib ファイルの中身は，次のようになっている． LaTeX でも使えていたような bib ファイルが，ほとんど修正することなく[†22] 使えることがわかるだろう．

```
@ARTICLE{Shannon_1948,
  author={Shannon, C. E.},
  journal={The Bell System Technical Journal},
  title={A mathematical theory of communication},
  year={1948},
  volume={27},
  number={3},
  pages={379-423},
  doi={10.1002/j.1538-7305.1948.tb01338.x}}

@book{Okumura_Kuroki_2023,
    author    = "奥村，晴彦 and 黒木，裕介",
    title     = "［改訂第9版］#LaTeX 美文書作成入門",
    publisher = "技術評論社",
    year      = 2023,
}
```

どうやら bib ファイル中に Typst のコマンドとして解釈できるもの（例えば `#LaTeX`）が入っていても，組版時に Typst のコマンドとして評価される訳ではないようである（[2] の出力結果がそうなっている）．

Typst で参考文献を出力するためには，

```
#bibliography("ref.bib")
```

と書く．出力結果は本稿の最後にある．ただし，`"ref.bib"`の部分は使いたい bib ファイルへの相対パスである． LaTeX で`\cite{foo}`とするところは，Typst では`@foo` とする．引用のための構文が相互参照のための構文と同じになっているので，当然キーの衝突は許されない．

なお，Typst では文献ファイルを YAML 形式で定義することもできる[†23]．しかし，LaTeX からの「移植」を考えたときには，わざわざ YAML に変換せずに bib ファイルのまま Typst に読み込ませたほうが簡単だろう．

文献[1] のタイトルが和文のカッコで囲まれているが，これは本稿が日本語設定になっているせいである．文献[2] の著者名が「と」で連結されているが，おそらく英語の“and”を機械的に訳した結果であろう．日本語の参考文献の表記法としては，例えば

†22 実際，Typst に移植するに当たって書き換えたのは，`\LaTeX` を `#LaTeX` に置き換えたことぐらいである．
†23 https://typst.app/docs/reference/model/bibliography/

奥村，黒木，『[改訂第 9 版] LaTeX 美文書作成入門』，技術評論社，2023.

のように，著者名を単純にカンマ（ないしそれに相当する記号）で連結する，という体裁が広く使われている[†24]．「と」が不恰好であれば，言語設定を参考文献だけ英語にする，つまり

```
#set text(lang: "en")
#bibliography( /* 省略 */ )
```

とする，という方法が考えられる．こうすれば，「と」が元の“and”に戻るので，不恰好さはいくらか低減される．とはいえ，この手法もワークアラウンドに過ぎないので，根本的には日本語向けの設定を探すほかないだろう．もっとも，Typst が参考文献のフォーマットのために使っているのは CSL ファイルであり，CSL ファイルは参考文献の書式を定義した XML ファイルなので，日本語向けフォーマットを定義するのは少し大変かもしれない．差し当たっては，日本語の学会論文向け汎用テンプレート[†25] に付属の CSL ファイルを流用するのが良いだろう．

思い出したので言及しておくと，Typst では英語のクオーテーションは単に'や"と書けばいい感じに判断してくれる．つまり，Alice and ‘Bob’ said that “this is good!”という文は，単に

```
Alice and 'Bob' said that "this is good!"
```

とすればいい．

3.9 箇条書き

箇条書きにもいくつかの種類がある．

- 番号なしの箇条書き．
- LaTeX では `itemize` 環境だった．

1. 番号ありの箇条書き．
2. LaTeX では `enumerate` 環境だった．

見出し 1 見出しありの箇条書き．

見出し 2 LaTeX では `description` 環境だった．

これは，次のように書けば実現される．

```
- 番号なしの箇条書き．
- #LaTeX では`itemize`環境だった．

+ 番号ありの箇条書き．
+ #LaTeX では`enumerate`環境だった．

/ 見出し1: 見出しありの箇条書き．
/ 見出し2: #LaTeX では`description`環境だった．
```

†24 例えば，JST による『参考文献の役割と書き方』(https://warp.ndl.go.jp/info:ndljp/pid/12003258/jipsti.jst.go.jp/sist/pdf/SIST_booklet2011.pdf)を参照．もちろん，学会や論文誌によって体裁は少しずつ違うかもしれない．

†25 https://typst.app/universe/package/jaconf

それぞれ，最初の記号（-，+，/）を変えることで実現される． Markdown の気分で，番号ありの箇条書きのマークとして 1.のように書いてしまうと，

```
1. foo
1. bar
```

と書いても

1. foo
1. bar

となり，番号がいずれも「1.」になってしまうので注意が必要である．これは，この構文は番号を手動でセットするための構文だからである[†26].

3.10 コメントアウト

最後にコメントアウト機能について触れておこう． LaTeX の % に相当するコメントアウトは//である． LaTeX の comment 環境に相当するコメントアウトは，/* ... */である．いずれもよくあるプログラミング言語と同様の構文になっている．

4 少し触ってみた感想

おそらく，よくある Markdown 程度の機能しか使わないのであれば，Typst でもそこそこの品質の文書を生成することができるだろう．その一方で，機能面（自由度とも言えるか）ではまだLaTeX に劣るのもまた事実であろう．また，日本語組版の品質はまだLaTeX のほうが優れている（本稿程度の簡単な文書ですら，既に微妙な点がいくつかある）．今後の発展に期待したい．

ちなみに，これを書いた本人は，個人使用であればまだLaTeX のほうが良いかなぁ……？と言っている．

謝辞

本稿執筆にあたり，編集部の Typst の先達によるコメントはたいへんに有益であった．「少しのことにも先達はあらまほしきことなり」（徒然草）．ここに感謝いたします．

5 参考文献

[1] C. E. Shannon, 「A mathematical theory of communication」, *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x.

[2] 奥村晴彦 と 黒木裕介, ［改訂第 9 版］*#LaTeX* 美文書作成入門. 技術評論社, 2023.

†26 https://typst.app/docs/reference/model/enum/

Tauri で移動記録アプリを作ってみた

文　編集部　北野 尚樹(puripuri2100)

1　はじめに

旅行やドライブなどの移動をしたときにその移動の記録を残しておきたいが、既存のアプリにデータを吸い取られるのはなんとなく嫌なので自力で何かツールを作りたいと、以前から漠然と考えていました。2024 年 10 月に、自分が以前から使っていた“Tauri”[†1] という GUI アプリ作成用フレームワークがモバイルアプリに対応したことを受け、この Tauri で移動記録アプリを作ってみることにしました。

この記事では、実装方法と出来上がったものについての概要と実際に使ってみた感想について述べたいと思います。なお、自分が使用している機材の都合で今回は Android 版の開発のみに限定しており、iPhone でのビルド方法などについては扱いません。

2　Tauri の概要と環境構築の方法

Tauri は GUI アプリ作成用のフレームワークです。表示する部分を JavaScript で書き、裏側で動く処理部分を Rust で書きます。JavaScript と Rust の間では Tauri が提供する橋渡し用のライブラリによってデータのやり取りが行われます。表示部分では React などの JavaScript による豊富な資産を使うことができ、データ処理部分では Rust による高速な処理が可能となる良さがあります。

プロジェクトを作成するために必要なものは次の通りです[†2]。

- Rust[†3]
- 好きな JavaScript のパッケージ管理システム
- Android Studio とそれに付随する SDK[†4]

準備ができたら、好きなパッケージ管理システムを使って好きなフレームワークと言語を使ってプロジェクトを構築します。今回の位置情報アプリは npm + React + TypeScript で作りました。

```
npm create tauri-app@latest
cd tauri-gps
npm install
```

†1　https://v2.tauri.app/
†2　https://v2.tauri.app/start/prerequisites/
†3　https://www.rust-lang.org/learn/get-started
†4　https://v2.tauri.app/start/prerequisites/#android

このままビルドするとスマホからアクセスできない旨のエラーが出るので、`src-tauri/tauri.conf.json` の `build.beforeDevCommand` の項目を `npm run dev -- --host` のように変更します。

環境変数の設定ができたら Android 用のバイナリのビルドを行います。

```
npm run tauri android init
npm run tauri android dev
```

これで Android Studio のエミュレータ機能を使ってアプリが立ち上がるはずです。

3 GPS を使って位置情報を取得する

Tauri のデフォルトでは位置情報を取得する機能はありませんが、プラグインを使うことで機能が解放されます。

Tauri のプラグインで公式のリポジトリ[†5] をグッとにらむと geolocation というやつが見つかります。このフォルダ内を探すと crates.io や doc.rs などのサイトにドキュメントが見つかります[†6][†7]。

これらのドキュメントを元に次のコマンドを実行してプラグインを追加します。

```
npm add @tauri-apps/plugin-geolocation
cd src-tauri
cargo add tauri-plugin-geolocation
```

次に Rust のファイルを変更します。`src-tauri/src/lib.rs` に次の一行を加えます。

```
#[cfg_attr(mobile, tauri::mobile_entry_point)]
pub fn run() {
    tauri::Builder::default()
        .plugin(tauri_plugin_shell::init())
+        .plugin(tauri_plugin_geolocation::init())
        .invoke_handler(tauri::generate_handler![])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

次に、設定ファイルでプラグインが提供する各機能を許可するかどうかを設定します。権限の許可は `src-tauri/capabilities/default.json` ファイルの“permissions”プロパティで行います。

```
{
  "$schema": "../gen/schemas/desktop-schema.json",
  "identifier": "default",
  "description": "Capability for the main window",
  "windows": ["main"],
  "permissions": [
    "core:default",
    "shell:allow-open",
+    "geolocation:allow-check-permissions",
+    "geolocation:allow-clear-permissions",
```

†5 https://github.com/tauri-apps/plugins-workspace/tree/v2/plugins
†6 https://crates.io/crates/tauri-plugin-geolocation https://docs.rs/tauri-plugin-geolocation/2.0.0/tauri_plugin_geolocation/index.html
†7 https://github.com/tauri-apps/plugins-workspace/tree/v2/plugins/geolocation

```
+    "geolocation:allow-get-current-position",
+    "geolocation:allow-request-permissions",
+    "geolocation:deny-watch-position",
+    "geolocation:deny-clear-watch"
  ]
}
```

最後に、React 側から JavaScript のライブラリを import して表示します。`src/App.tsx` ファイルを次のように変更することで、現在の位置情報を取得して表示するボタンを表示できるようになります。

```
import { useState } from "react";
import {
  checkPermissions,
  requestPermissions,
  getCurrentPosition,
  Position
} from '@tauri-apps/plugin-geolocation'
import "./App.css";

function App() {

  const [phonePos, setPhonePos] = useState<Position | null>(null);

 // 位置情報取得用の権限を与えてもらうようにユーザーにリクエストする
 // 権限がすでにある場合には現在の位置を取得してphonePos変数に与える
 async function getPos() {
  let permissions = await checkPermissions()
  if (
    permissions.location === 'prompt' ||
    permissions.location === 'prompt-with-rationale'
  ) {
    permissions = await requestPermissions(['location'])
  }

  if (permissions.location === 'granted') {
    const pos = await getCurrentPosition()
    setPhonePos(pos);
  }
}

  return (
    <div className="container">

      <button onClick={getPos}>位置情報を取得</button>
      <p>
        {phonePos ? <>({Math.round(phonePos.coords.latitude)},
        {Math.round(phonePos.coords.longitude)})</>: null}
      </p>

    </div>
  );
}
```

```
42
43 export default App;
```

4 地図を表示する

前節では緯度経度や高度までは取得できたのですが、実際に移動の記録を振り返ってみるときには地図上にその経路が表示されていることが望ましいです。今回は、地図を埋め込むための JavaScript ライブラリである Leaflet[†8] の React 用ラッパーである React Leaflet[†9] を使い、OpenStreetMap[†10] の地図を表示することにします。`MapContainer` で地図の大きさや拡大縮小の範囲などを指定し、`ChangeView` で更新を走らせ、`TileLayer` で表示される地図の種類を指定します。`Circle` では現在の位置に半径 40 の青い円を表示させるようにしています。

```
1  <MapContainer
2    style={{ width: "75vw", height: "60vh" }}
3    center={phonePos.coords}
4    zoom={14}
5    minZoom={5}
6    maxZoom={18}
7    scrollWheelZoom={true}
8  >
9    <ChangeView />
10   <TileLayer
11     attribution='&copy; <a href="https://www.openstreetmap.org/
       copyright">OpenStreetMap</a> contributors'
12     url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
13   />
14   <Circle
15     center={[phonePos.coords.latitude, phonePos.coords.longitude]}
16     pathOptions={{ fillColor: "blue" }}
17     radius={40}
18   />
19 </MapContainer>
```

5 作ってインストールしてみる

あとは平均時速を計算してみたり記録データをファイルで保存できるようにしたりといった機能を作り込んでいきます。パソコンとスマホを USB ケーブルで接続して実機を使ったデバッグモードを使ってみると、実機の挙動を見ながらデバッグできるのでかなり便利でした。

アプリが完成したらスマホにインストールしてみます。`npm run build` を行うと apk ファイルが出来上がります。これを Google Drive などを経由してスマホにダウンロードし、スマホ側から apk ファイルを開こうとするとインストールするかどうかの確認画面が出てくるので、同意をしてインストールします。このとき、apk ファイルに対して署名をしておく必要があるので気をつけてください[†11]。

†8 https://leafletjs.com/
†9 https://react-leaflet.js.org/
†10 https://www.openstreetmap.org/
†11 https://v2.tauri.app/ja/distribute/sign/android/

6　使ってみた

実際に春日から 3 学まで移動するところを記録してみた様子です（図 1）。3.2km を 15 分で移動できたことがわかるなど、かなり便利に使えています。現在位置の表示や記録の保存などもできるようになっています（図 2）。

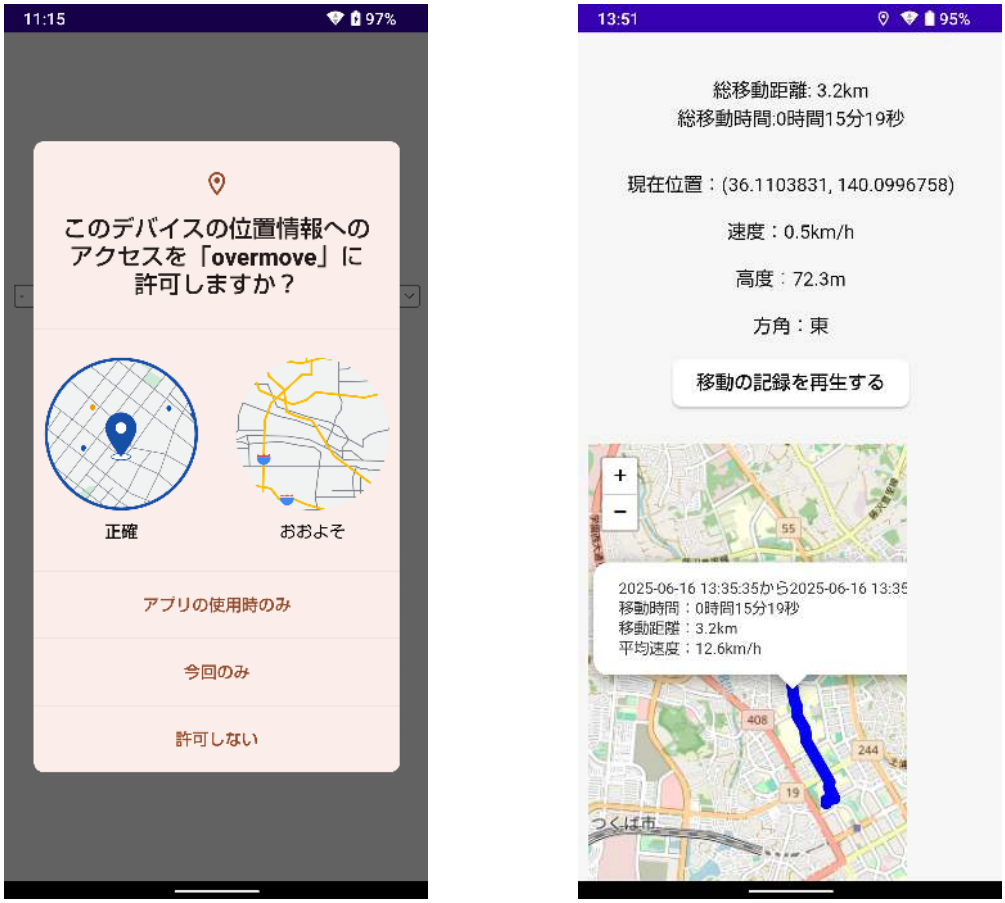


図 1: アプリを起動し、実際に移動を記録してみた様子

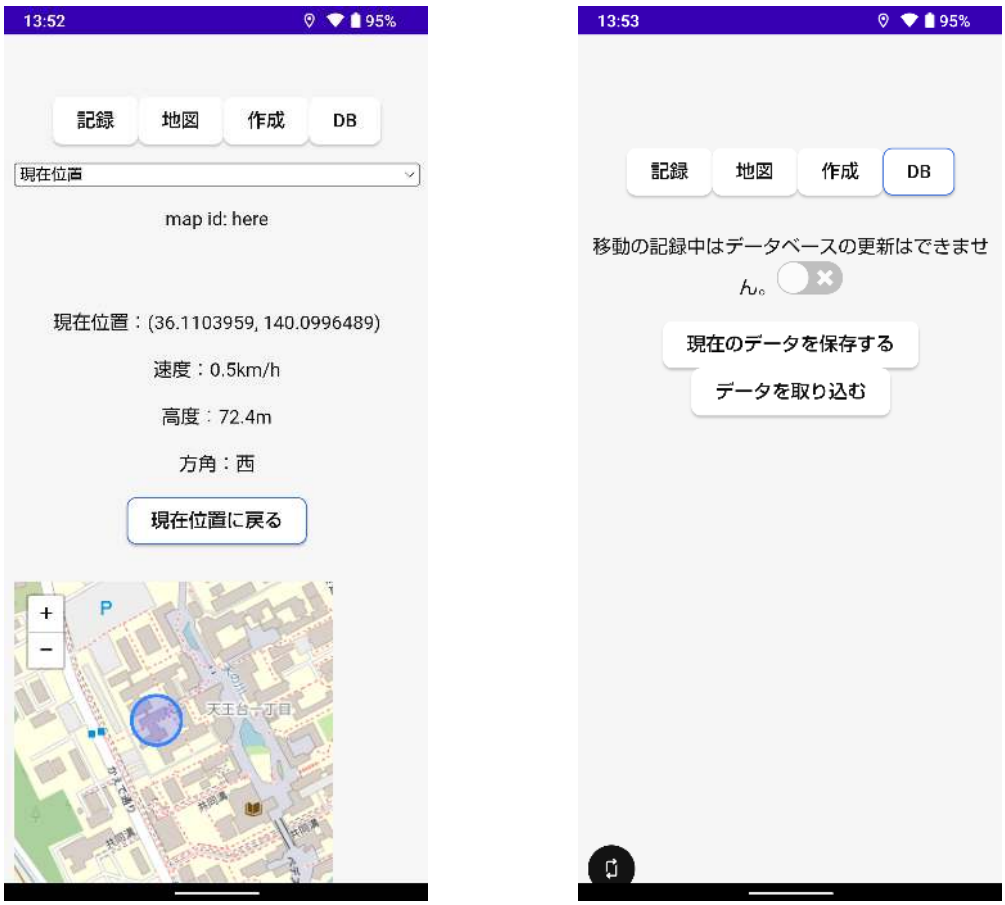


図 2: 現在位置の表示やデータの保存等の画面の様子

しかし、記録を常時取るためには常にアプリを起動し続けている必要があります。バックグラウンドではあまり GPS データを取ってくれず、直線的な記録になってしまいます（図 3）。Google Play で公開するときにバックグラウンドでも細かく GPS データを取るための権限を申請すれば解決できるらしいのですが、大変そうで手を出していません。

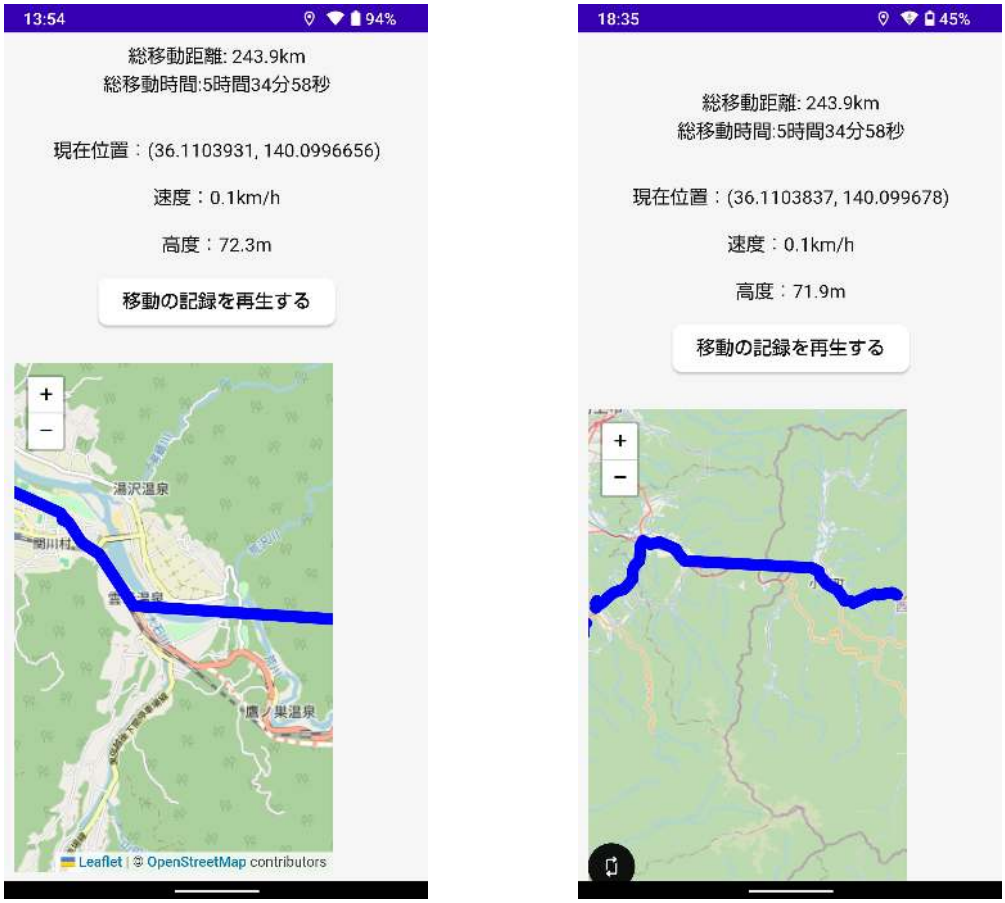


図 3: GPS 情報の取得間隔が開いてしまい、記録が直線的な移動になってしまった様子

7 おわりに

移動の様子を記録できるアプリをかなり簡単に作ることができました。コードは GitHub で公開している[†12] ので、ぜひ試してみてください。

UI を改善したり、スマホに依存しないデータの取り方を模索したりする必要がありそうなので、今後も暇なときに触っていこうと思います。

皆さんもぜひ Tauri を触ってみてください。単純に面白いですし、まだ機能に改善できる点も多いため、貢献チャンスだと思います。

†12 https://github.com/puripuri2100/overmove

Rust における動的ディスパッチと形式検証

文　編集部　lapla

1　はじめに

こん Rust〜（挨拶）．ところで Rust というプログラミング言語はある程度信頼性を担保しながらプログラミングを行えるようにすることが設計思想に比較的強く表れている言語です．この点において，Rust はコンパイラ自体が形式検証等の文脈における検査器的な営みをしている言語であるということができます．

しかしながら，動的ディスパッチを用いる場合には実行時に型情報が必要になるため，一見この設計思想と相反するように見えます．それだけでなく，動的ディスパッチを形式検証ツールが効果的に形式検証するためには実装上の工夫が必要です．

そこで本記事では，Rust における動的ディスパッチの実現方法とその形式検証手法について見ていきます．具体的には，最初に Rust のコンパイルプロセスについて概観し，その後に動的ディスパッチの仕組みを説明します．そして最後に Rust コードに出現する動的ディスパッチに対する形式検証手法について見ていきます．

2　Rust におけるコンパイル時中間表現

Rust はコンパイラ基盤として LLVM を採用していますが，コンパイル時にはいくつかの中間表現を経由してから LLVM IR に変換され，その後機械語に変換されます．本章では本題の前提としてそのプロセスを概観します．

最初に本章で紹介するプロセスを 1 枚の図にしたものを示しておきます：

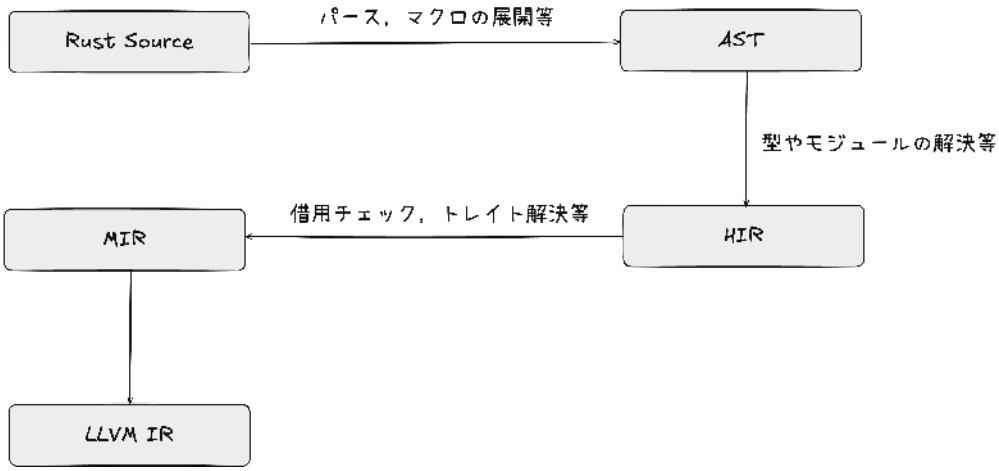


図 1: Rust のコンパイルプロセス

2.1 AST

パースされたソースコードを用いてまずは AST（Abstract Syntax Tree，抽象構文木）が作成されます．この時にマクロの展開等も行われます．

余談ですが，Rust におけるマクロはその定義方法により，`macro_rules!`を用いた宣言的マクロと proc-macro を用いた手続きマクロに分類することができます．このうち前者の宣言的マクロは新たなバージョン（提案段階では declarative macros 2.0 と呼ばれています）の導入が計画されています．この declarative macros 2.0 は，部分的衛生性を採用している従来の宣言的マクロに代わって，衛生的なマクロ（hygienic macros）を実現することを目指しています．具体的には，従来の宣言的マクロはローカル変数やラベル，`$crate` メタ変数については衛生的ですが，他の識別子については保証していないので，これをどうにかしたいという要請が根底にあるようです．

しかしながら，この機能についての RFC[†1] は 2016 年に作成されて以降依然として安定化されていない状態が続いて[†2] います．Rust にはこのように提案されて長らく安定化に至っていない機能が多く存在しており，個人的にはどうにかなると良いですねと思っています[†3]．

2.2 HIR

HIR（High-level Intermediate Representation）[†4] は AST から syntax sugar を取り除いたり，型チェックやモジュール解決を行った状態の中間表現です．

†1 https://rust-lang.github.io/rfcs/1584-macros.html
†2 https://github.com/rust-lang/rust/issues/39412
†3 これは何も全部安定化されれば良いと言っているのではなく，やるならやる，やらないならやらないを明確にすべきというだけの主張です．実際個人的には完全に衛生的なマクロって Rust に必要だろうかと思っています．
†4 https://rust-lang.github.io/rfcs/1191-hir.html

ここまでに紹介した AST や HIR も周辺ツールに用いられることがあります．例えば Rust の linter として有名な Clippy[†5] においては lint のパスとして `EarlyLintPass` と `LateLintPass` の 2 つが用意されており，前者は AST の段階に対して lint を行い，後者は HIR の段階に対して lint を行います．lint を実装する際には（lint を行うにあたって型情報が必要等の理由で）必要に応じてパスを選択します．また，Rust のコードフォーマッターである rustfmt[†6] は `syntex_syntax`[†7] というクレートによりエクスポートされた AST に対してフォーマットを行っているようです．

2.3 MIR

MIR（Mid-level Intermediate Representation）[†8] は CFG（Control Flow Graph）形式の表現です．借用チェックやトレイトの解決，Rust コンパイラが行う範囲での最適化，例えば定数伝播や不要な `Drop` の削除等が行われます．

最終的に MIR が LLVM IR に変換されてターゲットの機械語に落ちてくるといった具合です．MIR 自体は SSA（Static Single Assignment）ではありませんが，LLVM IR になる際に SSA になります．

3 Rust における動的ディスパッチ

Rust における動的ディスパッチはトレイトオブジェクトを用いて実現されています．一部の読者の方は動的ディスパッチの例として `&dyn Trait` や `Box<dyn Trait>`といった `dyn` キーワードを用いるような記法を読んだり書いたりしたことがあるかもしれません．実際，https://crates.io/上に公開されているクレートのダウンロード数上位 500 個について，37% のクレートが直接，70% が間接的にでも動的ディスパッチを用いているとの調査結果もあります [1].

トレイトオブジェクトとは，型に対する操作を行う際に実際の型が何であるかを意識せずに共通のトレイトを通じて操作できる仕組みのことを指します．ここで，Rust におけるトレイトとは 2003 年に Nathanael Schärli らによって提唱されたトレイト [2] とは似て非なる概念であることに注意が必要かもしれません．

デフォルトでは Rust はモノモーフィゼーションによりトレイトメソッドの呼び出しを静的に解決します（静的ディスパッチ）．これは実行時に余分なオーバーヘッドが乗らないものの，各型ごとに関数が作成されるので，コードサイズやコンパイル時間が増大するというトレードオフの関係が存在します．

一方それに対して動的ディスパッチは実行時にトレイトオブジェクトの型情報を参照してメソッドを解決するため，コードサイズやコンパイル時間は抑えられますが，実行時にオーバーヘッドが発生します．

†5 https://github.com/rust-lang/rust-clippy
†6 https://github.com/rust-lang/rustfmt
†7 https://crates.io/crates/syntex_syntax
†8 https://rust-lang.github.io/rfcs/1211-mir.html

動的ディスパッチが用いられる際には，内部的にはトレイトオブジェクトが「データ本体へのポインター」と「vtable へのポインター」を持っています．vtable とは仮想関数テーブルのことであり，本記事執筆時点で最新の安定版である Rust 1.87.0 においては `get_vtable` という関数[†9] において作成されているようです．C++等にも同じ名前の概念が存在しており，継承モデル等が異なるものの，達成したい目的は概ね同じです．

vtable には以下の情報が含まれます：

- 元の型のサイズ
- 元の型のアラインメント
- デストラクター
- トレイトで定義された各メソッドへの関数ポインター（が順に入っている）

トレイトオブジェクトには `Sized` トレイト[†10] は実装されないので値をコピー渡しすることはできません．そのため，トレイトオブジェクトはファットポインターと呼ばれる特殊なポインターの形式で渡されます[†11]．ファットポインターは雑に言えばポインターそのものとそれに付随するメタデータがセットになった構造を持つポインターのことを指します．

このように，Rust における動的ディスパッチはトレイトオブジェクトを通じて実行時に vtable を参照してメソッドを解決することにより実現されています．

4 動的ディスパッチの形式検証

さて，ここまでは Rust において動的ディスパッチがどのように内部的に実現されているのかを見てきました．ここからは，これをいかに形式検証できるのかということについて見ていきます．

Rust コードに対する形式検証ツールとして有名なものに Kani[†12] があります．Kani は Rust の MIR をベースとして，そのプログラムのメモリ安全性やオーバーフロー・アンダーフロー，assertion failure などの検証を行うことができます．

Kani それ自体は既存のツールには無かった `unsafe` ブロックを含むようなコードに対しても検証を行うことができるという特徴が最大の売りとされているように思いますが，Kani を用いて動的ディスパッチのような構造に対する検証を行うことができるという論文が Kani の開発者らによって発表 [1] されているので，今回は Kani をこの観点から見ていきたいと思います．

4.1 動的ディスパッチの検証時の課題

従来の Rust を対象とした検証ツールはダイナミックトレイトオブジェクトを完全にはサポートできていなかったり，MIR よりも下位の LLVM IR 等の言語に非依存な部分を対象にしていたりしたため，型情報の活用が困難でした．一方 Kani では MIR を対象としているた

†9 https://github.com/rust-lang/rust/blob/1.87.0/compiler/rustc_codegen_ssa/src/meth.rs#L103
†10 https://doc.rust-lang.org/std/marker/trait.Sized.html
†11 この点 C++の vtable のポインターはオブジェクト内にあるので単一のポインターでオブジェクトの全体を参照できます
†12 https://github.com/model-checking/kani

め，トレイトのセマンティクス情報を活用することによりダイナミックトレイトオブジェクトの検証を効率的に行うことができます．

実際には，Rust における動的ディスパッチを形式検証することには大きく次の課題があります：

- vtable のモデリング
  - vtable のレイアウトや動的ディスパッチのセマンティクスを正確にモデリングするのが難しい[†13]
- 関数ポインターの扱い
  - 動的ディスパッチでは関数ポインタが多用されるため，シンボリック実行やモデルチェックのパフォーマンスが悪化しやすい
- クロージャーの扱い
  - Rust におけるクロージャーも動的ディスパッチされるため，検証ツールはその挙動も正確にモデル化する必要がある

Kani はこれらの課題をクリアできるような設計がなされています．この論文では，以降で見ていく Kani の各ステップを表す図が次のように示されています：

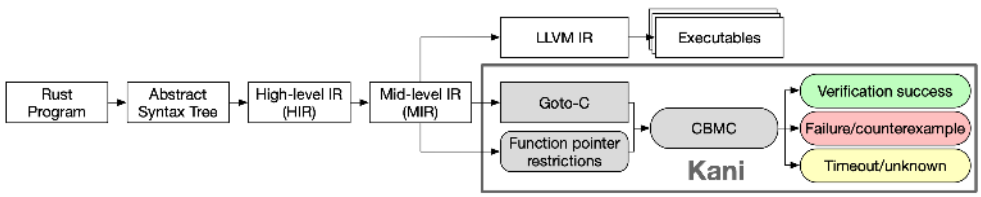


図 2: Kani の作用範囲を示した図

4.2 Kani の検証ステップ

Kani の検証ステップは次の 3 つに大別できます：

1. コンパイルして MIR を Goto-C 形式に変換する
2. シンボリック実行する
3. SAT ソルバーを用いて検証する

最初のステップでは kani-compiler クレート[†14] において，Rust のソースコードから MIR を作成し，それを Goto-C 形式[†15][†16] に変換します．最初に見たように MIR は SSA でなく，そこから変換される Goto-C 形式もまた SSA ではありません．

†13 というか公式に明確に規定されているわけではないので難しい
†14 https://github.com/model-checking/kani/tree/main/kani-compiler
†15 https://diffblue.github.io/cbmc/group__goto-programs.html
†16 ちなみに Goto-C はシリアライズする時に ELF としてバイナリにもできるし XML にもできるっぽいです．なんですかこれ

その後プログラムは CBMC[†17] というモデルチェッカーが扱える形式に変換されますが，このときには CBMC の `goto-instrument` と呼ばれるツールを用いて dead code elimination や最適化，必要なライブラリのリンクなどが行われます．そして `run_cbmc` という関数[†18] において CBMC 形式のプログラムへの変換が行われているようです．この時に関数ポインターの到達先を MIR に含まれる型情報を活用することにより制限することで，検証時の状態空間が爆発することを抑制しています．

そしてシンボリック実行の結果として，実行パスと検証すべき特性が含まれた単一の論理式を作成し，作成された論理式に対して CBMC が SAT ソルバーを呼んで検証を行います．この時利用できる SAT ソルバーはいくつか存在して，デフォルトでは MiniSat[†19] を用いるようですが，場面によっては異なるソルバーを用いた方が性能に顕著な差が出るようです[†20]．

実際に Kani は AWS が提供する Firecracker[†21][†22] の 16550A UART のシリアルデバイスや virtio-blk のブロックデバイスのパーサーの実装に対して検証を行い，従来の Rust 検証ツールよりも短時間で検証を終えることができることを示しています．具体的には，前者ではデバイスに対する動的ディスパッチを用いた read/write を行うそれぞれのメソッドの呼び出しを検証して関数ポインターの到達先制限により検証時間を短縮，後者では `std::io::Error` 等の標準ライブラリに含まれる型が動的ディスパッチを多く用いるので従来は検証に時間がかかっていたものの，Kani により検証時間を 15 倍以上改善できたとの報告がされています[1]．

5 参考文献

[1] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, “Verifying dynamic trait objects in rust,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 321–330. doi: 10.1145/3510457.3513031.

[2] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, “Traits: Composable Units of Behaviour,” in *ECOOP 2003 – Object-Oriented Programming*, L. Cardelli, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 248–274.

†17 https://www.cprover.org/cbmc/
†18 https://github.com/model-checking/kani/blob/kani-0.63.0/kani-driver/src/call_cbmc.rs#L82
†19 https://github.com/niklasso/minisat
†20 https://model-checking.github.io/kani-verifier-blog/2023/08/03/turbocharging-rust-code-verification.html
†21 https://github.com/firecracker-microvm/firecracker
†22 著者が Amazon の人間なので選ばれたのだと思います

Windows 使いは Linux の夢を見るか

文　編集部　にと

1　事の発端

皆さんの PC には何の OS が入っているでしょうか。おそらく多くの方が Windows と答えることでしょう。(macOS[†1]|Linux[†2] の方はすみません)

私が普段使っているメイン PC(デスクトップ)とサブ PC(Dell のノート[†3])にも例にもれず Windows が入っているのですが、開発をしているとどうしても UNIX[†4] ライクな環境が必要になる場面が出てきます。ある程度は WSL 2[†5] をこねれば何とかなるのですが、やはり素の Linux とは使い勝手が異なり[†6]、VM の上に立っている都合上余計にメモリを食うのでリソースの限られたノート PC[†7] では苦しいことがあります。そもそも、開発用途なら Windows である必要はなく、むしろ UNIX ライクな環境のほうが向いています。

そこで私も窓の外に飛び出す[†8] ことにしました。そうです、ノート PC に Linux を入れてしまえばすべて解決します。**WORD** 編集部では Linux の入った PC をメインに使っている人間が多く、もし困っても相談できる環境があったので思い切って OS を入れ替えることにしました。

2　いざ、Linux の世界へ

さて、ノート PC に Linux を入れるわけですが、Dell は自社の PC に Linux を入れることに気持ちがあるらしく、公式で Ubuntu をインストールする方法を紹介してくれています[†9]。これを活用しない手はないでしょう。8GB 以上の USB を用意し、指示に従って USB メモリにイメージを焼き、その USB メモリをノート PC に挿してインストール作業を行います。

†1　MacBook、ほしい
†2　GNU/Linux と言うべき、という説もある
†3　Dell Inspiron 13 5330
†4　macOS や Linux の祖先である(?)OS
†5　Windows Subsystem for Linux 2: Windows 上で Linux 環境を動作させるための機能
†6　Docker を使うときにいちいち Docker Desktop を起動する必要があったり、Windows 上のファイルへのアクセスが遅かったり等々
†7　ケチって 16GB にしたのがよくなかった、という説もある
†8　**WORD** 56 号参照
†9　https://www.dell.com/support/kbdoc/ja-jp/000131655/デルコンピュータに ubuntu-linux をインストールする方法

図 1: Ubuntu のインストール画面

思ったよりあっけなく Ubuntu が入ってしまいました。これでもう Windows とはお別れです。

3 Ubuntu に触れる

Ubuntu を日常的に使う日々が始まりました。もう WSL を使う必要はありません。ターミナルを開けば(当たり前ですが)Linux のコマンドが使えますし、Docker の起動に苦しむこともありません。Windows 時代よりも明らかに動作が快適で、バッテリーの持ちもよくなりました[†10]。日付と時間が常に上部のバーに表示されているのも地味に便利です。

若干心配していた指紋認証やタッチパッド[†11] も問題なく動き、Windows と変わらない、むしろそれ以上に使いやすい開発環境を構築することができました。

4 悪夢は突然やってくる

数日経って Ubuntu 環境にも慣れ始めたある日、それはやってきました。

いつものように Ubuntu を起動したところ、Ubuntu の起動画面の代わりに黒画面に白文字で何やらよくわからないメッセージが表示されています。これが噂のカーネルパニックというやつでしょうか。とりあえず適当なキーを叩いたりしてみますがうんともすんとも言いません。頭を抱えていると画面が切り替わり、次のようなメッセージが出ました。

†10 バッテリー持ちが悪くなる例もあるよう。
†11 スクロール速度には若干不満があったが…

```
BusyBox v1.36.1 (UBUNTU 1:1.36.1-6ubuntu3.1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

(initramfs) exit
Gave up waiting for root file system device. Common problems:
- Boot args (cat /proc/cmdline)
- Check rootdelay= (did the system wait long enough?)
- Missing modules (cat /proc/modules; ls /dev)
ALERT! UUID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX does not exist. Dropping to a shell!

BusyBox v1.36.1 (UBUNTU 1:1.36.1-6ubuntu3.1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

(initramfs) _
```

(initramfs)というプロンプトが表示されており、今度はキー入力が反応します。initramfs とは Linux が起動するときに最初に呼び出されるメモリ上に展開可能なファイルシステムのことです。ここからマウントしたいルートファイルシステム[†12]が呼び出されるのですが、これをうまく呼び出せない時にこのような画面になるようです。

とりあえずコマンドは打てるので update-initramfs や update-grub を試したりして、再起動してみます。

……..起動しました。

5　おや、Ubuntu のようすが…?

これで済めばよかったのですが、この時からうまく起動しなかったり、起動しても輝度が調整できなくなったり[†13]、解像度が狂ってとんでもなく字が小さくなったりと明らかに様子がおかしくなりました。

ひどいときにはターミナルすら出ずに BIOS エラーが表示されるとともにビー!!!!!ビー!!!!!!というけたたましい警告音を鳴らすようになってしまいました[†14]。完全に反抗期です。

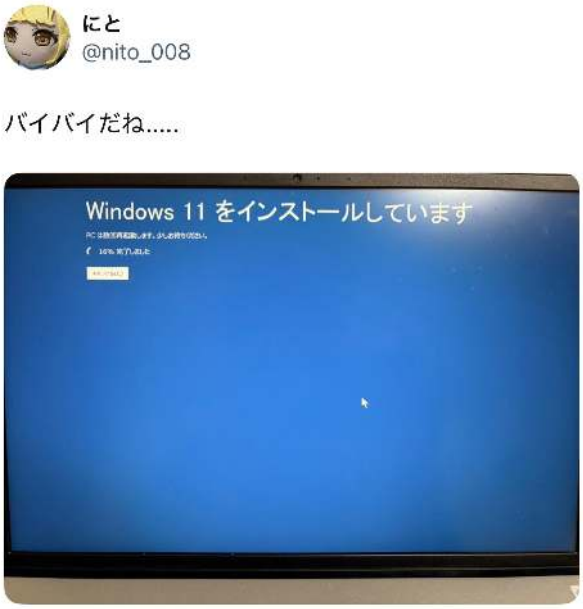
さすがに OS が起動しないのは苦しすぎるので Ubuntu を再インストールしようと思い、もう一度 Dell の Ubuntu インストール方法のページに行ったところ、次の項を見逃していたことに気づきました。

†12　ルートディレクトリ(/)が割り当てられたファイルシステムのこと
†13　輝度バーごと消失し、ボタンを押しても反応しない状態。別要因という説もある。
†14　自宅だったからよかったものの、講義中に鳴っていたら完全に終わっていた。

図 2: “デルコンピュータに ubuntu-linux をインストールする方法(https://www.dell.com/support/kbdoc/ja-jp/000131655/デルコンピュータに ubuntu-linux をインストールする方法)”より引用

そういえば私のノート PC は去年買ったばかりのものでした。BIOS を開いて確認すると `SATA Operation` に当たる項目は見つかりません。どうやら PC が新しすぎたようです。

6 そして、Windows へ



ハードの問題となるともうどうしようもありません。諦めて Windows を使うしか道はないようです。無念。

†15 あるまちゃん(https://www.miraclelinux.com/tech-blog/alma-chan)は可愛い

とはいえやはり Linux を開発環境として使いたいのは確かです。今度はヤフオクで落としてそのままになっているデスクトップに Alma Linux[†15] を入れるなり、新しく Linux 用ノートを買うなりして第 2 の Linux ライフを送ろうと思っています。

Auth0 で JWT 認証したいだけ

文　編集部　ゃー@reversed_R

やぁ、こんにちは。ゃーです。

6 月ですね。強いて言うなら 6 月 16 日です。~~記事の締め切りまであと 6 時間ですね。~~[†1]

さて、生きているとユーザー認証をしたい場面ってあると思います。ゃーも今作っているソフトウェアで必要でした。

1　ユーザークレデンシャルを持ちたい自信があるか?

ユーザーを認証するって言ったら最も最初に思いつくのは、ユーザーのパスワードを持っておくことです。

え？パスワード持ちたいっすか?

ゃーは嫌です。

もちろん誰かのクラウドは誰かのオンプレ理論と同様で、誰かは真面目に認証のためのパスワードを持つ必要はあります。

しかしパスワードを持つに当たって、というかパスワードをそのまま持ってしまっては DB が割られたら認証情報が見放題になってしまうので、適切にソルトやらペッパーやら付けまくってハッシュ化するなどしていく必要があります。

これを真面目にやってみることには価値がありますが、本当にセキュアなものを実現できるかはわからないですし、人様が作ってくれた信頼性のある(ということになっている)サービス[†2] の力を借りていきましょう。

その種の認証プラットフォームとしては、Auth0 とか Firebase Authentication、Amazon Cognito などあるでしょうが、今回は Auth0 を使うことにしました。

2　Auth0 のセットアップをする

Auth0 に登録します。

適当にテナントの ID やリージョン[†3] が割り振られます。`dev-xxxxxxxx` みたいになるようです。

よく知らんけれど、Database というのが認証されるユーザーのプールになるようですね。*Username-Password-Authentication* というデフォルトのデータベースが作られていますが、ユーザーリストを単体で持つなにかのサービスに対して 1 つずつ作ると正しいのでしょう。

ここで、Applications と APIs という概念があり、これはなんだろう、どこまで必要なんだろう、になります。

---

†1　人々が記事を書かなすぎて締め切りが 1 週間延びたらしいです。ゃーは間にあったんですがね。

†2　こういうサービス(SaaS)を IDaaS(あいだ〜ず)というらしいですね。

†3　別に日本から登録したからと言って JP っぽいリージョンになるわけではないようで、普通に us リージョンになりました。

2.1 APIs

アクセストークンにより認証、認可を行う必要がある保護されるべき APIサーバ(リソースサーバ)を指しています。Identifier として通常はドメインを指定するようです。後述しますがトークンを検証するために

- テナントのドメイン(`TENANT_ID.REGION.auth0.com` の形式)

を知っておく必要があります。

2.2 Applications

アクセストークンの発行を要求するクライアントアプリケーションのことを指すらしいです。どうやら、Native Application, Single Page Application, Regular Web Application, Machine to Machine Application から選べるようですね。今回はフロントは React を使うので SPA を選びました。トークンの発行には少なくとも

- テナントのドメイン(`TENANT_ID.REGION.auth0.com` の形式)
- Client ID
- audience(後述しますが、API サーバ側の Identifier のようです)

を持っておく必要があります。

3 アクセストークンを発行したい!!

とりあえずトークンが発行できることを確かめたいですね。Auth0 においては OAuth2.0 の JWT 形式が採用されているはずです。

なんと便利なことに、React に対しては auth0/auth0-react という NPM パッケージが提供されており、これを適当に使うことでログインやらの処理と UI が使えます。便利。

main.tsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import { Auth0Provider } from "@auth0/auth0-react";

ReactDOM.createRoot(document.getElementById("root") as HTMLElement).render(
  <React.StrictMode>
    <Auth0Provider
      domain={import.meta.env.VITE_AUTH0_DOMAIN}
      clientId={import.meta.env.VITE_AUTH0_CLIENT_ID}
      authorizationParams={{
        redirect_uri: window.location.origin,
        audience: import.meta.env.VITE_AUTH0_AUDIENCE,
      }}
    >
      <App />
    </Auth0Provider>
  </React.StrictMode>,
);
```

このように Auth0Provider というやつで適当に初期化してやり(見ての通り Auth0 のテナントのドメイン、クライアント ID、audience を環境変数から差し込んでいます)、

Me.tsx

```
import { useAuth0 } from "@auth0/auth0-react";
import { useState } from "react";
const Me = () => {
  const { user, isAuthenticated, isLoading, loginWithRedirect,
  getAccessTokenSilently } = useAuth0();
  const [token, setToken] = useState("");

  const getToken = async () => {
    const token = await getAccessTokenSilently();
    if (token) {
      setToken(token);
    }
  };

  getToken();

  if (isLoading) {
    return <p>Now loading ...</p>;
  }

  return isAuthenticated ? (
    <>
      <p>ユーザー名:{user?.name}</p>
      <p>メールアドレス:{user?.email}</p>
    </>
  ) : (
    <>
      <p>ユーザー認証が必要です</p>
      <button onClick={() => loginWithRedirect()}>Log In</button>
    </>
  );
};

export default Me;
```

このように、useAuth0()でパッケージが提供する様々を持ってきてやり、

- loginWithRedirect()を呼び出せばログイン画面に遷移
- getAccessTokenSilently()を呼び出せばログイン済み時にアクセストークンを必要に応じて取得
- user からユーザー関連情報(例えばユーザー名やメールアドレス)を取得

ができます。

3.1 ところで、トークン発行リクエストには何がレスポンスで返ってくるのか

いや〜、とても簡単で助かります。ありがたい話だ。

ところで、トークン発行リクエストに対して返ってくるのはどんなレスポンスなのでしょうか?

私、気になります!!見てみましょう。

```
{
  "access_token":"eyJxxxx..xxxx.xxxx.xxxx",
  "id_token":"eyJxxxx.eyJxxxx.xxxx",
  "scope":"openid profile email",
  "expires_in":86400,
  "token_type":"Bearer"
}
```

なんじゃこりゃ〜!!

`access_token` をよく見てみると、**JWT のはずなのに.が 4 つ**あります。

もちろん https://jwt.io でデコードもできません。泣きました。

強いて言うなら、ヘッダだけは以下のようにデコードできました。

```
{
  "alg": "dir",
  "enc": "A256GCM",
  "iss": "https://TENANT_ID.REGION.auth0.com/"
}
```

ヘッダって公開鍵の ID(kid)とかが含まれてるはず(あとアルゴリズムが `RS256` なはず)だが、なんか違うなあと思っておりました。

ここで JWT の形式について思い出そう

JWT は `header.payload.signature` の 3 部分からなる文字列です。

RFC7519[†4] で規定されているようです。

ヘッダ

ヘッダです。署名の方式などのメタデータが入っています。

データが入った JSON を Base64URL エンコードした値がヘッダになります。元々は下のような JSON だったわけです。

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

ペイロード

本体となるデータが載っています。

データが入った JSON を Base64URL エンコードした値がペイロードになります。元々は下のような JSON だったわけです。

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
```

[†4] https://datatracker.ietf.org/doc/html/rfc7519, OAuth2.0 での仕様は RFC9068(https://datatracker.ietf.org/doc/html/rfc9068)で規定。

```
  "iat": 1516239022
}
```

なお、予約語としていくつかのキーが定められています。

- `iss`: issuer。つまり、トークンの発行者。
- `sub`: subject。つまり、認証主体。認証されるユーザーの識別子。
- `aud`: audience。つまり、トークンによりアクセスできるリソースサーバ。場合によっては複数個指定できる。
- `iat`: issued at。つまり、トークンが発行された時刻(Auth0 では、UNIX timestamp)。
- `exp`: expiration time。つまり、トークンの有効期限。
- `jti`: JWT ID。つまり、トークンの ID。

署名

こうして得られたヘッダとペイロードそれぞれの Base64URL エンコードを.つなぎにしたものを、ヘッダで指定したアルゴリズムと発行者の秘密鍵を用いて署名付きハッシュを作ります。これを Base64URL エンコードしたものが署名部分になります。つまり、発行者の秘密鍵に対応する公開鍵で検証ができます。

なんで JWT じゃないものが降ってきているんだという問題

JWT じゃないものが降ってきている問題に戻りましょう。なんでやねん、と思いながら数時間クネクネし、インターネットを漁っても不透明なトークンみたいなよく分からんやつあって草とか思っていると、しばらく前に同じところで発狂していたぬっこくんが沼りました記事[†5] を出してくれました。

どうやら JWE[†6] というペイロードを暗号化済みの JWT みたいなもののようです。

RFC7516[†7] で規定されているそうな。

よく分からんのですが、 JWT って別に見られて問題になるような情報を載せるものではなく、改ざんが検知できるために信頼できる発行者からのトークンであることを検証できるもの、という理解をしているので、ペイロードを暗号化する意味があるのかは疑問です(誰かこの辺教えてください)。

ぬっこくんの記事曰く、audience を指定してやらないと JWE が返ってくるというのが Auth0 の仕様らしいです。

結構よく分からんけど、たしかにそのときのクライアントは上述のコードのようにはちゃんと audience を設定しておらず、設定したら普通の JWT が降ってくるようになりました。

なんなんこれ。

結局デコードすると

```
{
  "alg": "RS256",
  "typ": "at+jwt",
```

†5 https://mizuame.works/blog/2025-04-22/
†6 読み方がわからないけど、多分じょえ〜なんでしょうね。
†7 https://datatracker.ietf.org/doc/html/rfc7516

```
4     "kid": "xxxx"
5   }
6   .
7   {
8     "iss": "https://TENANT_ID.REGION.auth0.com/",
9     "sub": "google-oauth2|xxxx",
10    "aud": [
11      "https://MY.DOMAIN:PORT",
12      "https://TENANT_ID.REGION.auth0.com/userinfo"
13    ],
14    "iat": 1750051166,
15    "exp": 1750137566,
16    "scope": "openid profile email",
17    "jti": "xxxx",
18    "client_id": "xxxx"
19  }
```

が得られました。

アクセストークンに対して ID トークンって何すか

そういえば先程のトークン取得のレスポンスには `access_token` 以外に `id_token` というものがついていました。

これはなんじゃろう。

レスポンス再掲

```
1  {
2    "access_token":"eyJxxxx.eyJxxxx.xxxx",
3    "id_token":"eyJxxxx.eyJxxxx.xxxx",
4    "scope":"openid profile email",
5    "expires_in":86400,
6    "token_type":"Bearer"
7  }
```

これをデコードすると、

```
1   {
2     "alg": "RS256",
3     "typ": "JWT",
4     "kid": "xxxx"
5   }
6   .
7   {
8     "given_name": "ー",
9     "family_name": "や",
10    "nickname": "reversed_R",
11    "name": "やー",
12    "picture": "https://xxxx.googleusercontent.com/xxxx/xxxx",
13    "updated_at": "2025-06-15T07:03:22.892Z",
14    "email": "reversed_R@mail.domain",
15    "email_verified": true,
16    "iss": "https://TENANT_ID.REGION.auth0.com/",
17    "aud": "AUDIENCE_ID",
18    "sub": "google-oauth2|xxxx",
```

```
19   "iat": 1750051166,
20   "exp": 1750087166,
21   "sid": "xxxx",
22   "nonce": "xxxx"
23 }
```

ガッツリ個人のアイデンティティ情報(といっても認証に用いたメールアドレスに紐づいたものだが)が入っていますね。トークンを取得したクライアントアプリケーションが、誰としてログイン中なのかを提示できるような情報が含まれています。Gmail のアイコン画像のリンクなども入っていて、なるほど〜という感じです。

4　トークンを検証したい!!

そんなこんなでようやくアクセストークンが発行できました。
しかし、これを検証して認証を行うサーバがなくては意味がありません。
手順としては次のとおりです。

1. クライアントが `Authorization` ヘッダに取得したアクセストークン(JWT)を付与して HTTP リクエストをする。
2. API サーバが(認証を要するエンドポイントで)リクエストを受け取る。
3. `Authorization` ヘッダに付与されたアクセストークンの検証をする。
    - 発行者(issuer)を検証する。
        1. JWT のヘッダ部分をデコードすると、Auth0 が発行した JWT では `kid` が付与されている。
        2. API サーバはトークンの発行者を事前に知っているため、その公開鍵を取得する。
            - 公開鍵が不変である保証があるならば前もって行っておいても良いと思われます。
            - Auth0 の場合、`https://TENANT_ID.REGION.auth0.com/.well-known/jwks.json` から取得可能。
            - このとき取得できる鍵のセットを JWKS(JSON Web Key Set)というらしいです。
        3. 取得した JWKS から `kid` が一致する公開鍵を取得する。
        4. 取得した公開鍵で署名を検証する。
    - その他のペイロードを検証する。
        1. JWT のペイロード部分をデコードして含まれている値が正しいかを検証する。
            - `iss` が想定している発行者であるか。
            - `aud` に API サーバ自身が含まれているか。
            - `exp` の期限を過ぎていないか。
            - `npf` が含まれる場合、現在がその値以前でないか。

- などなど(もしかしたら用途によってはそれ以外を検証する必要もあるのかもしれません)。

実装している API サーバは Rust(Web API サーバ用の crate として Axum を使用)[†8] しています。

様々なエンドポイントに被せる必要性からミドルウェアの形式で実装しました。こんな感じで。

middlewares/user_auth.rs

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub(crate) struct Claims {
    pub iss: String,
    pub sub: String,
    pub aud: Vec<String>,
    pub iat: u64,
    pub exp: u64,
}

pub(crate) async fn jwt_auth(
    State(modules): State<Arc<Modules<DefaultRepositories>>>,
    mut request: Request,
    next: Next,
) -> Result<impl IntoResponse, AppError> {
    let authorization_header = request
        .headers()
        .get("Authorization")
        .ok_or(AppError::from(
            UserAuthError::MissingAuthorizationHeaderError.into_error_model(),
        ))?;
    let authorization = authorization_header.to_str().map_err(|e| {
        AppError::from(

            UserAuthError::InvalidAuthorizationHeaderError(e.to_string()).into_erro
        )
    })?;

    if !authorization.starts_with("Bearer ") {
        return Err(AppError::from(
            UserAuthError::InvalidAuthorizationHeaderError("Must Start With
            'Bearer'".to_string())
                .into_error_model(),
        ));
    }

    let jwt_token = authorization.trim_start_matches("Bearer ");

    match verify_access_token(
        jwt_token,
        &modules.config().app_url(),
```

[†8] かなり元ネタになっているそぽ祭りのリポジトリがこちらで https://github.com/sohosai/sos24-server、この実装自体は https://github.com/reversed-R/protalko-server に API サーバが、クライアントは驚いたことに(!!)Tauri でを使っていますが https://github.com/reversed-R/protalko-client にあります。メッセージングアプリにしたいんですがやること多すぎてワロタって感じですね。

```
        modules.config().auth0_url(),
        modules.repositories().user_auth_repository(),
    )
    .await
    {
        Ok(data) => {
            request.extensions_mut().insert();
            Ok(next.run(request).await)
        }
        Err(e) => Err(AppError::from(e.into_error_model())),
    }
}

async fn verify_access_token(
    token: &str,
    app_url: &str,
    auth_provider_url: &str,
    user_auth_repository: &impl UserAuthRepository,
) -> Result<TokenData<Claims>, UserAuthError> {
    let header = jsonwebtoken::decode_header(token)
        .map_err(|_| UserAuthError::InvalidTokenError("Failed to Decode
Header".to_string()))?;
    let kid = header.kid.ok_or(UserAuthError::InvalidTokenError(
        "Failed to Get Key Id".to_string(),
    ))?;

    let key = user_auth_repository
        .get_decoding_key_by_kid(kid)
        .await
        .map_err(|e| match e {
            UserAuthRepositoryError::KeyNotFoundError => {
                UserAuthError::InvalidTokenError("Key Not Found".to_string())
            }
            UserAuthRepositoryError::InternalError(s) =>
                UserAuthError::InternalError(s),
        })?;

    let mut validation = Validation::new(Algorithm::RS256);

    validation.validate_exp = true;
    validation.validate_nbf = false;
    validation.set_audience(&[app_url, &format!("{}/userinfo",
auth_provider_url)]);
    validation.set_issuer(&[&format!("{}/", auth_provider_url)]);
    validation.sub = None;

    let data = jsonwebtoken::decode(token, key.value(), &validation)
        .map_err(|_| UserAuthError::InvalidTokenError("Failed to Validate
Token".to_string()))?;

    Ok(data)
}
```

HTTP クライアントを立てて JWKS を取ってくる部分はこんな感じ。

infrastructure/user_auth.rs

```
pub struct Auth0UserAuthRepository {
    db: Auth0,
}

impl UserAuthRepository for Auth0UserAuthRepository {
    async fn get_decoding_key_by_kid(
        &self,
        kid: String,
    ) -> Result<DecodingKey, UserAuthRepositoryError> {
        let client = ClientBuilder::new()
            .timeout(Duration::from_secs(60))
            .build()
            .map_err(|_| {
                UserAuthRepositoryError::InternalError("Failed to Create HTTP
Client".to_string())
            })?;

        let jwks: JwkSet = client
            .get(self.db.auth0_jwks_url())
            .send()
            .await
            .map_err(|_| {
                UserAuthRepositoryError::InternalError("Failed to Send
Request".to_string())
            })?
            .json()
            .await
            .map_err(|_| {
                UserAuthRepositoryError::InternalError("Failed to Decode
Request".to_string())
            })?;

        let jwk: &jsonwebtoken::jwk::Jwk = match jwks.find(&kid) {
            Some(v) => Ok(v),
            None => Err(UserAuthRepositoryError::KeyNotFoundError),
        }?;

        let key = jsonwebtoken::DecodingKey::from_jwk(jwk)
            .map_err(|_| UserAuthRepositoryError::InternalError("Failed to Get
Key".to_string()))?;

        Ok(DecodingKey::new(key))
    }
}
```

Auth0 から取得したアクセストークンを、この API に対して `Authorization` ヘッダに Bearer として付与してリクエストをすると、無事検証に成功し認証が行えたのでした[†9]。

[†9] `jsonwebtoken::decode()`でちょっとコケました。コケるけどどんな種類のエラー(きっと enum なので)が返ってくるんだろうと docs.rs を見に行くと、戻り値の `Result` の `Err` のほうがないんですね。エラー無いやんけと思ったら、*type alias* らしい。そういう書き方もあるのか〜。

なお、JWT を検証しただけではユーザーを識別できずあくまで認可にとどまります。

実際にはユーザーを識別する認証が必要な場合が多いでしょうから、その際は sub などから認証主体の識別子を取得し、それをユーザーのキーの一種として DB に保管しておき、そのユーザーが可能なアクションを提供するとよいかと思います。

5 最後に

Auth0 での JWT 認証完全に理解した。

おおよその仕組み自体は知っているつもりでしたが、まあ実際に使って書いてみるほうが仕組みが分かりますね〜。

みなさんもぜひ、トークンを発行しまくり、トークンを検証しまくり、ユーザーを認証しまくりしてみてください。

ほんじゃまた〜。

結婚生活 1 年目振り返り

文　編集部　北野 尚樹(puripuri2100)

1　はじめに

去年の 7 月に結婚をしました。詳細については前号[†1] をお読みいただけると嬉しいです。

そろそろ 1 年が経ったということで、結婚してみてどうだったのか振り返ってみたいと思います。

2　結婚前と比べて変わったこと

2.1　良かったこと

生活が安定した

帰宅時間と起床時間がある程度相手に拘束されるようになり、生活リズムが狂う前に適切な時間に戻るようになった

栄養バランスの良い食事になった

相手の健康を意識して野菜が多く塩分が少なめの自炊を多くするようになり、ドカ食い気絶などの不健康な食事が減った

自律的な意思決定者が増えた

自分が忙しく生活に手が回らないときでも生活に必要な意思決定を行ってくれる存在が増えた

金融商品などの適用範囲が広がった

保険やクレジットカードなど様々な場面で出てくる「配偶者に限って可」という条件を満たすことができるようになった

日常生活が豊かになった

日常生活において外部からの刺激が定期的に発生するようになり、虚無の時間が無くなった

インターネットの時間が減った

インターネット意味ない

2.2　悪かったこと

パソコン力(ぢから)が落ちた

パソコンをしなくても刺激が得られるため、パソコンをしない日が増えた

†1　https://www.word-ac.net/post/2024/1123-word56/

3　日常生活はどうだったのか

夜に大学から帰って寝るまでの時間とたまにあるお互い暇な休日を一緒に過ごすのが主になり、それ以外の平日の昼間はそれぞれ大学などで過ごしているため共同生活でお互いの存在を意識して苦しむことはありませんでした。

4　義実家とのかかわりはどうだったのか

結婚して増えるものといえば義実家です。インターネット[†2]では日々怨嗟の声があふれていますが、実際には強い干渉が起きるようなイベント自体がありませんでした。たまにイベントに誘われたり帰省したりお土産があったりといった程度で、楽しい思い出になります。

5　総合的に見てどうだったのか

結婚してみてかなりよく、あっという間の一年でした。しかし、パソコンをやる時間が減ってしまったりとマイナスな点も少々ありました。今年からはパソコンを生活の一部にしていくことで持続可能なパソコン力を作っていきつつ、結婚生活 2 年目をやっていきたいです。

†2　Twitter（現 X）という名の地獄

古の趣味!?アマチュア無線の魅力

文　編集部　marunyann

1　概要

こんにちは。最近新たに WORD に加入しました。工学システム学類 2 年生の marunyann です。何の記事を書くか悩んでいましたが、よく考えればアマチュア無線をする wordian はあまりいないということに気が付きました。というわけで、ゆるふわ無線部員として 8 年目を迎える私が、進歩した電気通信に日ごろから触れていらっしゃる皆さんに、あえてふる〜い通信をする魅力をご紹介します。

2　アマチュア無線とは

アマチュア無線(アマチュア業務)とは、電波法令上は、「金銭上の利益のためでなく、もっぱら個人的な無線技術の興味によって行う自己訓練、通信及び技術的研究」とされています。つまり、仕事では使えない(近年なら使おうとすら思われなくなりつつある)ものですから、高級な電気通信を行っている皆さんの活動と比べたら、はるかに趣味のまま終わる可能性が高いです。しかし、「もっぱら個人的な無線技術の興味によって」という記述にある通り、やってみたいから、やってみる。ということがアマチュア無線の本質であり、あらゆる趣味の本質であると筆者は考えます。

3　アマチュア無線ならではの体験

3.1　聴覚受信

アマチュア無線の最大の特徴の一つは聴覚受信です。皆さんが日ごろ利用している高度な電気通信は、無線通信のものであっても、人間の聴力に頼って通信をすることはありません。根源的には 0 と 1 によって構成されるデジタルデータをアナログな電波に乗せて送信し、受け手側で、その電波からデジタルデータを復元し、しかも誤り訂正記号を用いた高度な信号処理(筆者にはよくわからない)によって、自動でデータの欠損や間違いを検出するというとてもとても難しい手法で行われています。つまり、電話であったとしても、LINE や Discord などの電話は、ノイズや音声の欠損によって、聞こえにくい、聞こえない部分ができることは基本的になく、受信する人間のスキルは特に関係ありません。しかし、アマチュア無線で利用される通信方法の大部分は、人間の聴覚に頼った受信方法です。機械が勝手にノイズと信号をはっきりと分離してくれることは基本的になく、アマチュア無線家の聴覚によってノイズと信号を分離し、さらに、完全に聞こえない部分を脳で補って通信する必要があります。この面倒な受信形体から発生した工夫がアマチュア無線らしさとして認知されています。

3.2 フォネティックコード

高度な電気通信には、誤り訂正符号なるものがあり、情報に冗長な情報を付加することで、符号化された情報に矛盾が生じた際に、データの欠損や誤りを自動的に検出することができます。つまり、通信に冗長性を付加しておくことで、ただ単に伝えたい情報だけを送信するよりも、通信の品質がだいぶ良くなるということが起こっています。同様のコンセプトは、アマチュア無線をはじめとする聴覚受信前提の通信方法にも存在しています。それがフォネティックコードです。フォネティックコードとは、文字を一文字ずつ送る際に、一文字一文字を特定の単語に置き換えて送る手法です。また、フォネティックコードを記載した表は、日本では欧文通話表と呼ばれます。表 1。例えば、ＷＯＲＤという文字列を送りたいとします。これを、「ダブリュー、オー、アール、ディー」などと送信すれば、まっとうに受信してくれる可能性は非常に低いでしょう。なぜなら、聴覚受信においては、信号とノイズを人間の耳で分離するため、ノイズ交じりの音が相手に聞こえているためです。どういうことかというと、「ダブザ゛ァァァァァァオー、ザ゛ァァァァルル、ｄザ゛ァァァァァァァァァァァァァァァァ」のように聞こえていることがあるということです。しかし、フォネティックコードを用い、「ウイスキー、オスカー、ロメオ、デルタ」と送れば、多少のノイズが入ったとしても「ウザ゛ァァァァァキー、オｓザ゛ァァァァァァァァァァメオ、Delザ゛ァァァァァァァ」のように聞こえていたとしても、一応なんの文字が送られてきたかは理解することができます。これがフォネティックコードの仕組みです。

文字	通話表コード
A	ALFA
B	BRAVO
C	CHARLIE
D	DELTA
E	ECHO
F	FOXTROT
G	GOLF
H	HOTEL
I	INDIA

文字	通話表コード
J	JULIETT
K	KILO
L	LIMA
M	MIKE
N	NOVEMBER
O	OSCAR
P	PAPA
Q	QUEBEC
R	ROMEO

文字	通話表コード
S	SIERRA
T	TANGO
U	UNIFORM
V	VICTOR
W	WHISKEY
X	X-RAY
Y	YANKEE
Z	ZULU

表 1: 欧文通話表 (Phonetic Code)

3.3 モールス信号

アマチュア無線の醍醐味といえば、古き良き伝統的なモールス信号です。モールス信号は、表 2 に示す方法で、文字を短点と長点で表す手法です。

文字	モールス符号
A	・－
B	－・・・
C	－・－・
D	－・・
E	・
F	・・－・
G	－－・
H	・・・・
I	・・
J	・－－－
K	－・－
L	・－・・
M	－－

文字	モールス符号
N	－・
O	－－－
P	・－－・
Q	－－・－
R	・－・
S	・・・
T	－
U	・・－
V	・・・－
W	・－－
X	－・・－
Y	－・－－
Z	－－・・

文字	モールス符号
0	－－－－－
1	・－－－－
2	・・－－－
3	・・・－－
4	・・・・－
5	・・・・・
6	－・・・・
7	－－・・・
8	－－－・・
9	－－－－・

表 2: 欧文・数字モールス符号

一見すると、符号が 2 種類しかないため、2 進数による情報伝達と同様になっていて、string 型で文字列を送っているようなものに感じるかもしれません。しかし、実はそんなに単純ではないのです。表をよく見ると、一文字当たりの長さが決まっていません。つまり、デジタルデータ通信よろしくすべての符号を「・－－－－－・－・－・・－・－・－－－・・－・・・・」のように連続的に送信しようものなら、どこで文字が切れているのか見当もつきません。これでは使えませんね。そこで、通常文字と文字の間に短点 1 個分、単語と単語の間に長点 1 個分の何も送信しない時間を設けます。ちょうど、

「・－－　－－－　・－・　－・・　　－・－・　－－－　・・　－・　・・・」

といった具合です。これなら、「WORD COINS」と言っていることがわかるでしょう。このため、筆者はむしろ三進数的な文字伝達ではないかと感じています。さて、モールス信号には、ある一つの強みがあります。それは、比較的雑音に強いということ。電話タイプの通

信方法であっても、単語の聞こえない部分を補完したり、フォネティックコードを使用することで、一般的でない単語を何とか伝達したりすることはできるものの、たった二種類の音と空白を聞き分けられればいいモールス信号は圧倒的にノイズに強いのです。筆者が、ヨーロッパ(具体的にはイギリスやフィンランドなど)のアマチュア無線局と交信したときは、モールス信号が多かったなぁという程度の優位性があります。さて、ここからは、関係あるようで関係ない話ですが、アマチュア無線をやっていると、「モールス信号わかるの？じゃあこれわかる？」といったような無茶ぶりを振られる。という妄想をするものですが、実際には、人生そう甘くはありません。モールス信号を理解しない人が作った日本のコンテンツにおけるモールス信号を利用したイースターエッグはたいてい和文モールス信号で構成されており、覚える文字が多い、そのことからそもそも符号が長いものが含まれている、そもそもあんま使わん。といった理由で覚えていないのでわからないのです。悲しいなぁ。

3.4 アンテナ自作

アンテナは比較的簡単に自作が可能です。アンテナの要求性能はとてもシンプルで、端子から、電線部分まで、全部くっついたときに、出したい周波数での特性インピーダンスがだいたい 50Ω であれば OK です。

缶のアンテナ、缶テナ

アマチュア無線家は様々なものをアンテナに変えてしまいます。(昔は、どこどこのゴルフクラブはよく飛ぶ。という話をしていて、よく聞けば、アンテナにしたときに電波がよく飛ぶという意味だった。というような恐ろしい話があったそうです。)その一つが空き缶です。このようなアンテナは、アマチュア無線家のあいだでは、缶テナとよばれ親しまれている、技術的にはグランドプレーンアンテナと呼ばるアンテナの一種です。 図 1 にグランドプレーンアンテナの概念図を示します。CAD で 5 分ぐらいで作った 3D モデルなので、見苦しいですが、おおむねこうです。

- 給電部から、4 分の 1 波長のエレメントを垂直に伸ばします。
- 給電部から水平に 4 分の 1 波長のエレメントを何本か伸ばします。

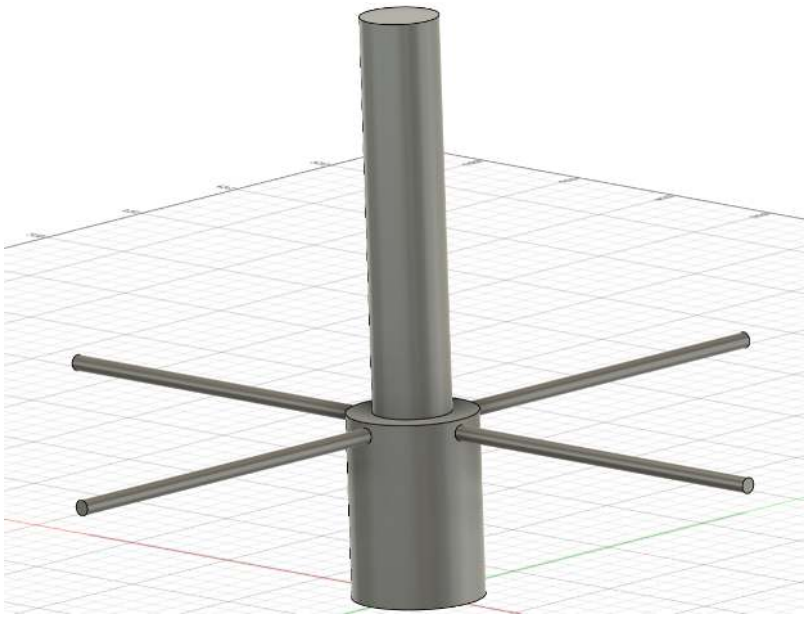


図 1: グランドプレーンアンテナの概念図

このうち、水平に伸ばしたエレメント(ラジアルともよばれます)は、給電部の cold とつながっています。これによって、仮想的に、電位がゼロの平面を作ることができます。これによって、下側にも同じ長さの垂直なエレメントがあるかのようにふるまいます。よって、グランドプレーンアンテナは、中心に給電部がある全体の長さが 2 分の 1 波長のアンテナかのようにふるまいます。[1] 二分の一波長の長さのアンテナは、もっとも、効率的に定在波をつくってくれるので、これによって、インピーダンスの整合ができ、電波を飛ばすことができるわけです。ここで、アマチュア無線の世界で人気のある周波数帯の一つ、430MHz 帯の代表的な周波数 433MHz の波長は約 692mm ですから、4 分の 1 波長となると、173mm です。このことから、大体の長さがこれに近い缶を用意すれば、垂直なエレメントを缶で作り、水平なエレメントは、適当に、金属製の棒なんかをつけておけば、何とかグランドプレーンアンテナを構築することができます。よく使われている缶は、500ml ビール缶(高さ約 167mm)、400mlZONe の缶(高さ 163mm)モンスターエナジー 350ml 缶(高さ 162mm)などです。電気科の苦悩[2] に詳しい作り方が書かれています。

ワイヤーダイポールアンテナ

ワイヤーダイポールアンテナを制作する際は、多くの場合、バランと呼ばれる給電部をあらかじめ購入してきて、そこに、 4 分の 1 波長の電線を二つくっつけてダイポールアンテナを作ります。このときに、10m 以上の線を測って切らなければならないので、たいていの無線部には、長いメジャーがあります。給電によって焼き切れてしまうような軟弱なワイヤーを使用しなければ、本質的には、金属棒を使用したダイポールアンテナと同様に使えます。

バラン部分と、超長い電線さえ用意すれば、あとは、左右から碍子と、不導体のワイヤーで引っ張るだけなので、とても簡単に施工することができ、波長が、40m に達する 7MHz 以下の周波数で広く使われています。

4 必要なもの

情報通信が発達した今だからこそ、ユニークな体験ができることが魅力のアマチュア無線は、残念ながら始めるために多くのコストがかかります。ここでは、アマチュア無線を始めるために必要なものについて解説します。

4.1 免許

アマチュア無線では、かなり大きな電力を電波として放出します。何の知識もない人がこのような設備を使用すれば、数多くの業務通信を妨害してしまうかもしれません。このため、もし、全人類が無秩序にアマチュア無線を始めてしまえば大変なことが起こるので、アマチュア無線を始めるためには免許が必要となっています。それでは、アマチュア無線を始めるために必要な二種類の免許について解説します。

無線従事者免許 アマチュア無線技士

アマチュア無線ができると認められている人間に対して交付される免許です。取得すると免許証、つまり、免許を証する書類をもらえます。これは携帯する前提のものになります。小学生でも取れるとされている 4 級から、合格するために、大学生並みの電磁気学の知識が必要とされている 1 級まであり、一度取得すると一生使えます。写真付きの公的身分証明書にもなるので、それを目当てに 4 級アマチュア無線技士を取得する人もいるそうです。

無線局免許

アマチュア無線従事者の免許証を持っている人間と、アマチュア無線設備の組み合わせであるところのアマチュア無線局に対して、交付される免許です。取得すると無線局免許状がもらえます。これは、免許の有効期限が 5 年しかなく、しかも年々更新の期間が厳しくなっています。

4.2 無線機

アマチュア無線といえば、無線機を自作。そんなイメージを持つ人はたくさんいるでしょう。実際、筆者も無線機を自作するのか問われたこともたくさんあり、さらには、無線機を自作している前提で話しかけてくる人さえいます。が、近年では、アマチュア無線局からの混信などを減らすためなのか、それとも一般に混信が増えているのか、それとも単に技術が向上したからか、不要発射の量がとても厳しく規制されています。このため、近年無線機を自作する人は非常にまれで、私も作ろうと思ったこともありません。お手軽に近場と交信するならスマホほど、図 3 本格的に日本全国、あわよくば世界中と交信するためには、パソコンほどの値段をはらって無線機を購入する必要があります。図 2

図 2: 本格的なアマチュア無線に使う無線機。この無線機は初心者向け と主張する人もいるが、1 台 1 台がパソコンみたいな値段だし、用途で考えれば十分 本格派だと思う。母校のアマチュア無線部の備品。母校で撮影

図 3: お手軽な無線機。多くの人類がトランシーバーと聞いて、 想像する形。日本語ではハンディ機、英語では Walkie-Talkie と呼ばれるイメージ。 筆者の私物。母校で撮影。

4.3 アンテナ

なぜか法規上は空中線と書かれています。読んで字のごとく、最小構成のアンテナは、本当に空中に電線を引いたような見た目をしています。もちろん、市販のアンテナのほうが高性能ですが、先述のとおりアンテナは割と簡単に自作が可能なので、作ってみるのもいいでしょう。

5 あとがき

いかがだったでしょうか。アマチュア無線には、ここに書き切れないぐらい、なんなら私にも理解しきれないほどのたくさんの魅力がありますが、筆者は必修単位と格闘する工シス2年生なので、古い無線通信ならではの特徴、始めるために必要なもの、この二つだけにしぼって、書かせていただきました。面白かったと思っていただけたら幸いです。

6 参考文献

[1] 編集部,「アマチュア無線のアンテナを作る本[HF/50Mhz 編]」. [Online]. 入手先: https://shop.cqpub.co.jp/hanbai/books/16/16471/16471.pdf

[2] 電気科の苦悩,「魔材アンテナ(カンテナ)を作ろう」. [Online]. 入手先: https://kstak.hatenablog.com/entry/2018/09/16/190624

WORD Typst 化計画

文　編集部　Ryoga (@Ryoga_exe)、おかし (@oka4shi)

1　はじめに

WORD 57 号（今号）より、試験的に WORD は Typst を用いて組版されることとなりました。今回は、その背景や Typst 化における取り組みについてご紹介しようと思います。

2　背景

2.1　従来の LaTeX の組版の問題点

従来のテンプレートは LaTeX 組版のもの[†1] でしたが、編集部内から以下のような課題が指摘されていました。

- ビルド時間が長い
- 依存関係が巨大
- プレビューの即時性がない
- Markdown の変換機能による破壊

特にビルド時間の長さは大きなボトルネックでした。記事の校正フローにおいて、何度も修正を加えることが発生し、そのたびに長い GitHub Actions が実行されることが多々ありました。

Typst への移行を決めた最大の動機もここにあります。Typst は差分ビルドによる高速コンパイルに加え、プラグイン機構で柔軟なカスタマイズが可能なため、これらの課題を一挙に解消できると判断しました。

また、プレビューの即時性についてもメリットが大きいです。`typst watch` コマンドを使用すればホットリロードを実現でき、記事執筆の体験が飛躍的に向上することが見込まれます。

2.2　Typst が成熟してきた

2025 年 2 月に Typst は v0.13.0 がリリースされました。そこでいくつかの機能が追加され、LaTeX による WORD の記事テンプレートを Typst で再現できるほどの表現力を持つようになりました。

†1　https://github.com/WORD-COINS/word-template

2.3 Markdown での執筆環境

従来の LaTeX 組版テンプレートでは、Pandoc による Markdown での執筆がサポートされていました。

WORD56 号「プロポーズされたら WORD 号」では 15 記事中 12 記事が Markdown で執筆されており、Markdown 執筆のニーズの高さがわかります。しかし、Pandoc による Markdown 執筆では、画像まわりで破壊が起きることが多々あり、修正にかなりの労力が要されることがありました。結局 LaTeX の記法を持ち出して修正する必要があり、Markdown のシンプルな記法というメリットをあまり享受できない状況だったのです。

Typst では Markdown ライクな構文をネイティブで提供している上、`cmarker`[†2] というプラグインにより、Pandoc なしでの Markdown 執筆に対応しているため、このワークフローを更にシンプルにできます。

3 Typst 化における取り組み

基本的に WORD 伝統の組版のデザインを Typst で再現しました。[†3] GitHub Actions の詳細については次節で述べます。

Typst 組版については、 WORD55 号[†4] の「大学のレポート、Typst で書いてみませんか」や WORD 入学祝い号 2025[†5] の「Coins 新歓パンフを支える仕組み」をご覧ください。

ここでは、Markdown での執筆環境のサポートについて書きます。

前述した通り、Markdown での執筆環境では、 `cmarker`[†6] というプラグインにより実現されています。改ページなどの Markdown では表現できないような組版は、HTML 記法をオーバーライドする機能により対応させました。

実際の Markdown テンプレートでは以下に示す Typst が使用されています。

```
#import "/template/article.typ": article
#import "@preview/cmarker:0.1.5"
#import "@preview/mitex:0.2.5": mitex

#show: article.with(
  title: "記事を執筆しよう",
  author: "情報 太郎",
)

#cmarker.render(
  read("main.md"),
  math: mitex,
  scope: (
    image: (path, alt: none, ..args) => figure(
      image(path, alt: alt, ..args),
      caption: alt,
```

†2 https://typst.app/universe/package/cmarker/
†3 https://github.com/WORD-COINS/word-template-typst
†4 https://www.word-ac.net/post/2024/0801-word55/
†5 Web 版は記事執筆時の 2025 年 6 月にはまだ出ていないようです
†6 https://typst.app/universe/package/cmarker/

```
    ),
  ),
  html: (
    pagebreak: ("void", _ => pagebreak()),
    h: ("void", attrs => {
      h(int(attrs.value) * 1pt)
    }),
    v: ("void", attrs => v(int(attrs.value) * 1pt)),
    img: ("void", attrs => figure(
      image(
        attrs.src,
        width: eval(attrs.at("width", default: "auto")),
        height: eval(attrs.at("height", default: "auto")),
      ),
      caption: attrs.alt
    ))
  )
)
```

このように書けば、`<pagebreak />` で改ページがされ、`<h value="10">` などによってスペーシングを調整することができるようになります。

また、`img` タグをオーバーライドすることにより、画像周りの処理を簡潔に表現できるようにしました。`<img alt="キャプション" src="ファイル名" width="50%">` と書くと `50%` という文字列が Typst に渡されて評価され、画像の幅の長さとして処理されます。

4 GitHub Actions について

4.1 従来の運用

WORD では、LaTeX を使用していた従来から GitHub を用いて記事の管理を行っていました。各個人が書いた記事を GitHub に push し、レビュー（赤入れ）を行ったのち、それら一つにまとめて WORD を発行するという流れです。GitHub Actions を用いて、記事をビルドして LaTeX のコードから PDF を生成する過程を自動化していました。

しかし、記事の自動ビルドに時間がかかる、ビルド結果の PDF の確認が手間といった問題がありました。

4.2 Typst 化に伴う変更点

これまでは、LaTeX をビルドするためにコンテナを使用していました[†7]。しかし、Typst は一つのバイナリで完結し、プラグインも実行時に自動でダウンロードされるためコンテナを使用する必要がありません。

このコンテナの廃止と、LaTeX を Typst に置き換えたことにより、Workflow の実行にかかる時間を大幅に少なくすることができました。これまで一つの記事のビルドに 2 分程度かかっていたものが、10 秒程度にまで短縮されました。これによって、「記事の自動ビルドに時間がかかる」という問題が解決されました。

†7 https://github.com/WORD-COINS/latex-build

4.3 Artifact のプレビュー

最初は、ビルドされた PDF を WORD 編集部でホストしているオブジェクトストレージ (MinIO)にアップロードしていました。その後、トークンの管理が問題となり GitHub Actions の Artifact に保存するように変更されました。しかし、GitHub Actions の Artifact は、必ず ZIP ファイルに圧縮されてしまうという制約と、Workflow の実行結果の画面からしかアクセスできないという問題があります。通常この仕様が問題になることはあまりありませんが、生成された PDF を確認するという用途にはそれが手間になっていました。

そこで、GitHub Actions の Artifact をブラウザ上でプレビューするためのサイトを作成し、そのリンクをプルリクエストにコメントするようにしました。これによって「ビルド結果の PDF の確認が手間」という問題も解決することができました。

図 1: 実際のコメントの例。このプレビューのリンクを踏むと自動でダウンロードが始まり、ブラウザ上で PDF が開く

リンクのコメント

GitHub Actions の Workflow の中ではデフォルトで GitHub CLI（gh コマンド）が使用できるため、プルリクエストにコメントを付ける際はそれが便利です。既存の Workflow に以下のようなコマンドを追加して、コメントを付けるようにしました。

```
gh pr comment "${{ github.event.pull_request.html_url }}" --body "${message}" --edit-last --create-if-none
```

`--edit-last` オプションと`--create-if-none` オプションを使用することで、コメントが存在しない場合は新たにコメントを投稿しつつ、存在する場合は前のコメントを編集するようにしています。

プレビューサイトの作成

ZIP ファイルをブラウザ上で展開して中の PDF ファイルをブラウザでプレビューするためのサイトを作成しました[†8]。

まず、GitHub の Artifact の URL からは直接ファイルがダウンロードできるわけではなく、その URL からリダイレクトされる先のオブジェクトストレージの署名付き URL からダウンロードする必要があります。これは、ログイン状態で URL にアクセスするか、GitHub API を

†8 https://github.com/WORD-COINS/artifact-previewer

使用することで取得できます。今回は、GitHub API にリクエストして署名付き URL を取得するための API を作成し、Cloudflare Workers 上に配置しました。

そして、取得した署名付き URL から ZIP ファイルをダウンロードし、ブラウザ上で展開して PDF をプレビューするためのフロントエンドを作成しました。これは、Vite + Vanilla-TS で構成されています。JavaScript(TypeScript)を使用して ZIP ファイルを展開するためには一工夫が必要です。ZIP ファイルでよく使われる圧縮形式である Deflate は、ブラウザの標準 API である `DecompressionStream` を使用して展開することができます。しかし、ZIP ファイル自体にはメタデータが含まれていたり、複数のファイルをまとめる機能があったりするため、それらを解釈するコードは自身で実装するかライブラリを使用する必要があります。

ZIP ファイルの展開

ZIP ファイルの解析にはライブラリを使用することもできますが、`DecomppressionStream` を使うようなライブラリには決定的なものが見つからなかったため、独自に実装しました[†9]。

ZIP ファイルフォーマットの大まかな構造は以下のようになっています。

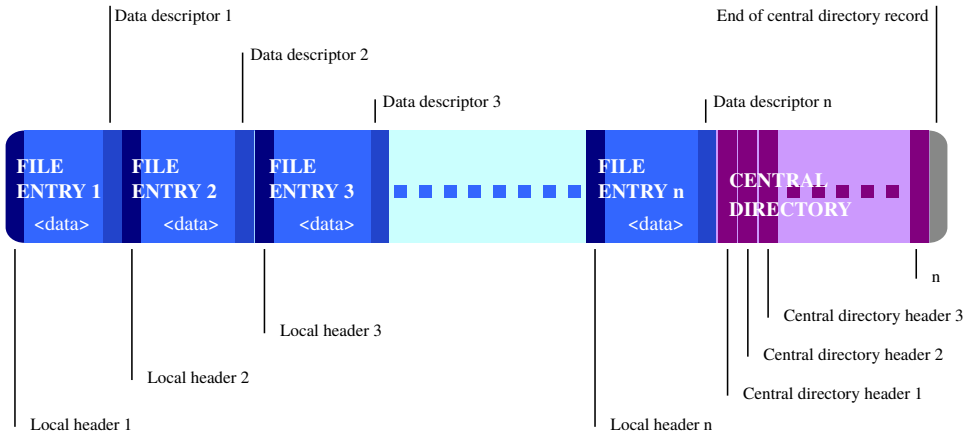


図 2: ZIP ファイルフォーマットの構造[†10]

これを展開するためには、まずファイルの終端から前に向かって探索し、End of central directory record(EOCD)の始まりを示す `0x504B0506` を探します。ここには Central directory の数や Central directory の開始位置へのオフセットなどが含まれています。これを参考にして Central directory を読み取ります。

各 Central directory には、フォルダ構造やファイル名、更新日時やサイズ、圧縮方法などのメタデータが含まれています。ここからファイルの Local header の開始位置と圧縮状態でのサイズ、それから圧縮方法を取得しておきます。

†9 `DecompressionStream` を使わないものは定番があるようでしたが、全モダンブラウザで使える標準 API があるならそれを使いたいじゃないですか。

†10 「© Stkl 2016. “ZIPformat.svg”. https://commons.wikimedia.org/wiki/File:ZIPformat.svg. (ライセンス URI: https://creativecommons.org/licenses/by-sa/4.0/)」を改変して作成。この図も CC BY-SA 4.0 でライセンスされています。

次に、Local header の開始位置を元に Local header を読み取ります。Local header の後に実際の圧縮されたデータが格納されているため、その位置から圧縮状態でのサイズ分だけ内容を取り出します。その後、圧縮方法の数字をもとに適切な解凍アルゴリズムを選択して展開します。この数字は、例えば、`0` であれば非圧縮（データそのまま）、`8` であれば Deflate 圧縮が使用されているといった具合になっています。大抵の ZIP ファイルでは Deflate 圧縮が使用されているため、多くの場合は `DecompressionStream` を使用して展開することができます。GitHub Actions の Artifact でも例に漏れず Deflate 圧縮が使用されているため、これを利用して展開しています。

実際に Deflate 圧縮された部分を取得して解凍する処理を TypeScript で実装した場合のコードは以下のようになります。`fileDataOffset` には開始位置、`compressedSize` には圧縮状態でのサイズ、`buf` には ZIP ファイルのバイナリデータが格納されているとします。

```
// ファイルを取得
const fileData = new Uint8Array(
    buf,
    fileDataOffset,
    compressedSize,
);

// Uint8Array<ArrayBuffer>をBlobに変換
const ZIPBlob = new Blob([fileData]);

// BlobをReadableStreamに変換
const stream: ReadableStream<Uint8Array> = ZIPBlob.stream();

// Deflate圧縮を解凍するDecompressionStreamを作成
const compressedStream = stream.pipeThrough(
    new DecompressionStream("deflate-raw"),
);
const res = new Response(compressedStream);

// ファイルを解凍し、Blobとして取得
const decompressedFile = await res.blob();
```

これで ZIP ファイルの展開が完了します。`decompressedFile` には解凍されたファイルの Blob が格納されているため、あとは自由に扱うことができます。例えば、PDF ファイルであれば、`URL.createObjectURL()` で Blob にアクセスできるような URL を生成してその URL をブラウザで開くことでプレビューすることができます。

5 おわりに

Typst への移行はまだ試験運用段階であり、実際今号の執筆のなかでいくつかテンプレートの問題点も見つかりました。しかし、ほとんどはすぐに修正可能であるものであり、何より原稿の執筆から校正までのサイクルは大幅に短縮できる見込みがあります。

新しい紙面の裏側で進む技術刷新にもぜひご注目ください。

私的な近況報告

文　編集部　すい

皆さんはじめまして、情報科学類一年のすいと申します。このたび Wordian になりまして、初めて記事を執筆させていただいております。 何を書けばいいのか少し迷っているのですが、技術系の話は今のところサッパリなので、とりあえずわたくしの近況をつづってみたいと思います（社会的なテーマであれよ）。

1　KINKYO

1.1　自転車で事故る

入学早々、下り坂にて猛スピードで木に突っ込み、救急車で搬送されてしまいました。 幸い大事には至らなかったのですが、顔に目立つ傷ができてしまいました。小ショックです(悲)。 さらに、怪我の影響で新歓に参加しそびれ、各種の機会を損失しました。

ただ、悪い事ばかりではありません（と、思いたい）。この事故を経て身体の大切さを実感出来ました。頭だけで考え事をしていても、 いざという時に身体を動かせないと意味がない。体育があってその上に知育がある、という今まで逃避し続けてきた現実を直視するいい(?)機会になりました。

1.2　たのしいつくば生活

つくばでの生活は毎日がエブリデイでエキサイティングです。もちろん落ち込んだりもしますけど、基本的に元気に過ごせてます。元気が一番ですね！

一人暮らしもこの上なく自由で楽しいです。好きなときに遊ぶ生活をすれば、あっという間に 1 日が過ぎます。 動画を見たりゲームをしたり、映画とかアニメ、読書とか音楽とかとかとか。 どれだけ暇になれば、 1 日が 24 時間で足りるのでしょうか？

1.3　優しい、真面目な人が多くて驚く

大学に来て一番意外だったことはこれです（意外といったら失礼ですが）。というのも、高校時代、私は大学生に対してあまり良い印象を抱いていませんでした。 大学に入ったら自慢争いや学歴トークの波に揉まれるのだろうと覚悟をしていたのですが、いざ入学してみると、そういうものを好まない方が多くて安心しました！ また、本気で学問的な思索を好んでいる方がめっちゃ多く、感動しました。

1.4　買い物にハマる

つくばに移住してからというもの、買い物を楽しいと思うようになりました。今まで未経験だったネット通販にも手を出しました。Amazon ってスゴイ。 なぜ急にお買い物にハマっ

たのか、理由は自分でもわかりませんが、買い物の自由度が増したのは一因でしょう。ただし、買い物の頻度が高まると散財をしてしまい危険なので、 そこは気を付けたいと思います。

2 おわりに

いかがでしたか？(まとめブログ) 私は今自分で記事を読み返してみて、内容のあまりのとりとめのなさに頭を抱えています。面白いテーマはどのようにすれば思い浮かぶのでしょうか… とにかく、近況という自分語りは、まだ私には面白く扱えないテーマだったようです。そう考えると、私小説を書いてる作家さんってすごいですね。

英語には、“Oversharing“という言葉があるといいます。これは、SNS などでプライバシーに関わる情報を過度に公開する行為を指します。 この概念は、自分語りの一部に含まれるでしょう。何が言いたいのかというと、自分語りは基本やめたほうがいいです(1 敗)。それではまたいつか…;)。

Interop Tokyo 2025 に行ったログ

AK47

1　ごあいさつ

はじめまして、AK47 と申します。

まだ正式な**Wordian**でないため、今回は寄稿という体裁を取っております。次号以降で、正式な執筆者としてお会いしましょう！

軽く自己紹介をしておきますと、人文・文化学群人文学類所属の B1 で、情報学をこねくり回して人文学に応用しようという野望を抱いている者です。Secure SHell ではない方の SSH と呼ばれる文科省指定の怪しい高等学校の出身です。高校時代は C++、Flutter、Go、Rust、TypeScript あたりを触っており、DiscordBot やスマフォアプリ、ヴィジュアルノベルゲームエンジンの開発を行っていました。近々、それらについての記事も執筆するかもしれません。乞うご期待！

さて、自己紹介はこのくらいとして、本題に参りましょう。

今回私が執筆するのは、先日幕張で開催されていた、**Interop Tokyo 2025** というイベントへ赴いたというログです。~~幕張は千葉県なのに *Tokyo* なんですね！~~

ゃーさんや間瀬 bb さんを始めとした他の**Wordian**の先輩方も数人行ってらっしゃったそうですが、あいにくゃーさんとは別日で、間瀬 bb さんとは入れ違いとなってしまった為、会えずじまいでした……。そういう訳で、私の視点からのみかつ非公式のイベントレポートとなってしまうことをご了承ください。

では、早速レポートに移ります。

2　ほんへ

Interop の会場たる幕張メッセの最寄駅、千葉県は海浜幕張駅に着いたのは、日本標準時で午前 09 時 46 分。快晴の空の下、QVC~~福島~~を通り過ぎながら、千葉ロッテマリーンズのユニフォームを着たプロ野球選手にまじまじと見つめられつつ、メッセへと着弾。

いきなりトレンドマイクロがお出迎えしてくれました。~~ウイルスバスターバスターが必要と n 回……~~

受付を済ませると、東京五輪マスコットキャラクターのミライトワとソメイティもお出迎え。なぜここにいるのかはよくわかりませんでしたが、あれからもう 4 年の月日が経ったのですね。

さて、満を持して中へ入りますと、**ドーン！**

感動！Interop さんブースデカいのね！

さて、どこから行こうかと会場案内図を凝視した結果、IT に強そうな囚人たちが衆人監視を受けながらネットワークを運用している、**ShowNet** へ行くことに。

ShowNet Team Member の欄に、本学の**くろー**さん(菊田さん)の名前もありますね。お忙しかったようで、ご尊顔を拝謁できなかったのですが、後程 Twitter を拝見した所、Most Valuable STM 2025 として表彰されたそうです。おめでとうございます！

さて、所は変わって、IPA や Internet #sym.times.big Space Summit 等を見て回りました。時折強そうな人と名刺を交換したりもしまして、とても有意義な交流ができたと思います。

IPA ブースでは、「800Gbps でパケットを送って機械をいじめています！」という話を聞きました。私の家の有線 LAN は 1Gbps なので、その 800 倍ですね。意味がわかりません。

Sky 株式会社のブースです。お馴染みの藤原竜也氏ですね。本学は 2B 棟食堂が「Sky CENTER TERRACE」となっており、そちらでも有名だと思います。また、先日に東大の五月祭へお邪魔した時も、Sky 後援の部屋がありました。どうやら色々な所にいらっしゃるようです。

……何？最近世間を騒がせている筑波大学のニュースに関して、俺たちには知る権利があるって？……いえ、これ以上はやめておきましょう。

撮影した会場内の写真はこのくらいで、後は某ルートのおかげで VIP 扱いになった為、VIP ラウンジへ行ったりしました。タダのコーヒーは美味しかったです。

かくて会場を脱出し、イオンモールにて「サイゼで喜ぶ AK47」をやったりしましたが、後々になって「Cloudflare ブースに行きそびれた！」ということに気づきました。いつもお世話になっており、一言くらい挨拶したかっただけに、反省。来年は行きたいですね。

最後に幕張豊砂駅で撮影した快速列車が通過する様子の写真を添えて。なお、この後は南流山駅からあきばエクスプレスに乗り換え、6 限の東洋思想へ出席する為大学へと向かうのでした。

だが、AK47 は知らなかった。そこで、Interop の荷物に加えてゴミケからモニタを 2 台担いで実家へ持って帰る地獄が待っていることを……。

←To Be Continued？

これにて、イベントレポートの本編は終了となります。あまり中身のあるレポートではありませんでしたが(出せない神レポより出すゴミレポ！)、Interop の中ってこんな感じだよ！という雰囲気が伝わっていれば幸いです。

3　来年、行こうかな？という人への注意点

- 学生お断りの雰囲気がそこそこあります

今年はホームページに「学生歓迎」と書きながら、実際の来場登録にあたって、法人名が必須でした。このことが槍玉に挙げられ、Twitter では「学生お断りではないか！」という論調も散見されます。

ちなみに、私はどの法人の所属でもないので、法人名の欄に「なし」と書きました。すると、来場者カードに「**なし**、AK47」と書かれて印刷されました……。法人所属の同行者に爆笑されました。

もし来年行きたいよ！という方がいらっしゃいましたら、そうしたことは留意した上で行くと良いと思います。

- お堅いスーツである必要はなさそうです

今年は、ホームページに「ビジネスの場ですので、それに相応しい服装でお越しください」と書いてありました。私はそれを真に受けてスーツで行ったのですが、会場へ入ると意外に私服の方も多く、スーツである必要性はあまりなかったように思われました。

あまりにもオタクを全面に出したフルグラフィック T シャツ等は流石に厳しいですが、落ち着いた服装であれば、私服でも良いかもしれません。

- 名刺は持っていきましょう

例によってホームページには「受付で名刺を 1 枚回収します」とありますが、今年は回収されませんでした。しかしながら、名刺の交換をして人脈を作っておくと、世界が狭いことに気づくこともできますので、名刺は 20 枚程度持って行くことをオススメします。

4　おわりに

いかがでしたか？この記事が役に立ったかどうかの正解は 1 年後ですが、Interop に興味を持って頂ければ幸いです。

では、また次号でお会いしましょう！

CodeQL を使ってみよう

文　編集部　いわんこ

1　まえがき

こんにちは！いわんこ(X: @iwancof_weakptr)と申します。普段は、天井を眺めるか pwn をしています。

wordian なのか wordian でないのか自分でも確定していなかったのですが、意味不明なので記事を書いて確定させたいと思います。

2　CodeQL とは？

CodeQL とは、GitHub が開発した**プログラミング言語に対する**クエリ言語です。ここではまず導入として簡単な例を提示し、その解説を行います。どんなツールかが雰囲気だけでも伝わると嬉しいです。

ちゃんとした解説を次以降のセクションで行いますので、導入を面白いと思ってくださる方がいらっしゃいましたら、ぜひご一読ください！

さて、さっそく次の C 言語のコードを、後述するクエリで処理してみます。

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("it works!");
5 }
```

```
1 import cpp
2
3 from FunctionCall fc
4 where fc.getTarget().getName() = "printf"
5 select fc, fc.getEnclosingFunction(), fc.getLocation()
```

すると、このような結果を得ることができます。

```
1 Compiling query plan for /home/iwancof/WorkSpace/CodeQL/article/queries/printf.ql.
2 [1/1 comp 8.9s] Compiled /home/iwancof/WorkSpace/CodeQL/article/queries/printf.ql.
3 Starting evaluation of qlsample/article/queries/printf.ql.
4 Evaluation completed (243ms).
5 |       fc       | col1 |                              col2                              |
6 +----------------+------+----------------------------------------------------------------+
7 | call to printf | main | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:4:5:4:10 |
8 Shutting down query evaluator.
```

クエリの内容を解説しつつ、結果を考察してみましょう。

from や where、select など、クエリ言語らしいキーワードが散見されます。ざっくりとした解説になりますが...

```
1  from FunctionCall fc
```

まず、from 句で変数の定義を行います。ここでは、FunctionCall という型を持つ fc という実体を定義したと考えてください。FunctionCall とは、名前の通り**関数呼び出し**を表しており、fc は関数呼び出しという**事象**が入ると思ってください。

```
1  where fc.getTarget().getName() = "printf"
```

次に、where 句で fc をフィルタしています。型にはメソッドが存在して、getTarget や getName がそれに該当します。雰囲気的に、「関数呼び出しの内、そのターゲット（呼び出される関数）の名前が"printf"のもの」をフィルタしていそうです。

```
1  select fc, fc.getEnclosingFunction(), fc.getLocation()
```

最後に、select 句を用いてフィルタした fc や fc に付随する情報を表示します。getEnclosingFunction はその関数呼び出しを行っている関数を返し、getLocation は関数呼び出しが行われた場所を返します。

結果を再掲します。

```
1 |       fc       | col1 |                              col2                              |
2 +----------------+------+----------------------------------------------------------------+
3 | call to printf | main | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:4:5:4:10 |
```

３つのカラムが返ってきていて、それぞれ fc, col1, col2 という名前がついています。fc は"printf"という関数呼び出し実体が入っています。

col1 は fc.getEnclosingFunction()の結果が入っていて、実際に printf を呼んでいる main 関数の実体を取得しています。

col2 は fc.getLocation()の結果で、その呼び出しが行われている場所を一意に特定する文字列が返ってきます。この場合、printf.c という名前でテストしていたので、そのファイル名と行番号などを得ることができています。

この例では、CodeQL を使ってコード自体をデータベースとして解析し、関数呼び出しの実体とそれを含む関数やその場所を取得しました。CodeQL ではより高度なクエリを書くことで、更に多くの情報をソースコードから得ることができます。また、C 言語だけでなく、数多くの言語を扱うことができ、それぞれ専用のライブラリが存在しています。詳しくは公式のドキュメントを参照してください。

次のセクションから、環境構築、実行方法、クエリの基礎、クエリの応用、原理を紹介します。もちろん公式の包括的なドキュメントも存在し、いずれはそちらを参考にするときが来ると思います。しかし、ドキュメントの読解に際し、CodeQL への最低限の理解が要求され、そのための日本語の情報が乏しいということでこの記事を執筆しています。ただ、筆者

の CodeQL への理解が甘いことに起因し、記事を読んでいるだけでは理解できない部分も多々あると思います。実際に手を動かし自分で検証を行ったり、もしくは X や Discord(@Iwancof_ptr) で質問していただいても結構です！連絡お待ちしています。

また、記事の作成には x86_64 Linux を使用しています。それ以外の環境に関しては詳しくありませんが、MSVC や C# .NET にも対応していることから、多くの環境で使えると思います。

3 環境構築

CodeQL はパッケージマネージャで配布されていないため、GitHub からバイナリを取ってきます。

https://github.com/github/codeql/tags

ここから、適当に latest を取ってきて展開し、展開したディレクトリに PATH を通します。`codeql` というコマンドが動かせたら準備完了です。

VSCode に CodeQL 用の拡張があり、結果を GUI で扱いたい方は併せて導入をおすすめします。LSP もありますので、Vim を使っている方も安心してください。

4 実行方法

では実際にクエリを書き、実行してみましょう。

クエリの実行には大きく次のステップを踏む必要があります。

1. ターゲットビルド・データベースの作成
2. クエリの作成
3. クエリの実行

4.1 ターゲットビルド・データベースの作成

まず解析対象のプログラムを作成し、そのプログラムから CodeQL 用のデータベースを作成します。このデータベースにはソースコードのすべての情報が含まれることになります。

例で使ったプログラムを流用しましょう。次のプログラムを `printf.c` として保存してください。

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("it works!");
5 }
```

次に、`codeql` コマンドでデータベースを作成します。

```
1 $ codeql database create ./db --language=cpp --command="clang printf.c
  -o printf"
```

`./db` で出力先を指定します。 `--command` で指定したコマンドを実行し、ビルドをトラップして情報を収集します。

内部的には `codeql/cpp/tools/linux64/extractor` がこれを行っています。使用しているコンパイラが `extractor` と互換か否かは、ドキュメントで確認することができます。

https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/

ただし普通に嘘をついているので気をつけてください。

例えば、一部の GCC の拡張構文は `extractor` が対応していません。`database create` する際はなんのエラーも表示しませんが、該当ファイルのデータベースだけ作成されないためかなり見つけづらい不具合が発生します。その場合、`db/log/build-tracer.log` を見ることでエラーを確認することができます。

具体的にこれが問題になるのは、Linux Kernel のデータベースを作成する際です。`clang-17` 以外のコンパイラを使うと `fs/select.c` など一部のファイルがデータベースに登録されなかったりします。

小さいプログラムで試す分には `gcc` でも良いですが、本格的に使う場合は、少し古めの `clang` を使うと良いです。

4.2 クエリの作成

実際にクエリを作成します。

先ほど例示したクエリを再掲します。

```
1 import cpp
2
3 from FunctionCall fc
4 where fc.getTarget().getName() = "printf"
5 select fc, fc.getEnclosingFunction(), fc.getLocation()
```

これを、`search_printf.ql` 等、名前をつけて保存してください。

4.3 クエリの実行

クエリの実行前に、内部で使っているパッケージ(`import cpp`) をローカルに入れます。

次の内容を、`qlpack.yaml` として保存してください。

```
1 name: sample
2 version: 0.1.0
3 library: true
4
5 dependencies:
6   codeql/cpp-all: "*"
```

次に、`qlpack.yaml` と同じディレクトリで、

```
1 codeql pack install
```

とすることで、指定したパッケージをいれることができます。

最後に、指定したクエリと対象のデータベースを指定し、クエリを実行します。

```
1 codeql query run -j0 ./search_printf.ql -d ./db
```

以上が、指定したクエリを指定したデータベースで実行する方法になります。これ以外にも、CSV 出力や Graphvis で描画可能な dot ファイルを出力する方法もありますが、ここでは割愛します。

5 クエリの基礎

CodeQL のクエリは Prolog の方言と言われています。筆者は Prolog や Datalog に詳しくないため、ここでは、最低限クエリを書く具体的な方法だけを説明します。

クエリは基本的に次のような構造をしています。

```
1 from ValueType value
2 where cond
3 select value
```

5.1 `from` 句

`from` 句では変数の宣言ができます。冒頭の例では `FunctionCall` という型の変数 `fc` を宣言していました。ここで重要なのは、`fc` を宣言した瞬間、**すべての関数呼び出し**が `fc` に入っているということです。

`FunctionCall fc` と宣言した時点で、実際にすべての関数呼び出しを集合として評価し、その一つ一つが `fc` に入ると捉えてください。

例えば、「すべての関数を列挙する」というクエリを考えてみましょう。冒頭の例では関数コールを列挙するために `FunctionCall` を使っていました。今回は関数を列挙したいので、関数を表す型である `Function` を使います。

```
1 import cpp
2
3 from Function func
4 select func, func.getLocation()
```

先程の説明通りにこれを解釈すると、`Function func` とした時点ですべての**関数という実体**が集合としてまとめられ、その一つ一つが `func` に入ると予想されます。実行してみましょう。

```
| func | col1 |
+---+---+
| __overflow | file:///usr/include/stdio.h:950:12:950:21 |
| __uflow | file:///usr/include/stdio.h:949:12:949:18 |
 (中略)
| rename | file:///usr/include/stdio.h:160:12:160:17 |
| remove | file:///usr/include/stdio.h:158:12:158:17 |
| main | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:3:5:3:8 |
```

実行すると、100 行ぐらいの長めの結果が帰ってきます。これは、ソースコードの先頭で `#includ <stdio.h>` と書いた際に、`stdio.h` に定義されている関数を CodeQL がまとめて関数として認識するからです。

`Function` はデータベース内のすべての関数を扱うため、`where` 句で条件を絞らないと、すぐに結果が膨大になってしまいます。また、より大きなデータベースでは、out-of-memory として問題になることもあります。条件を忘れているか緩すぎるせいで結果が大きくなっていますので、クエリを見直してみてください。

`from` 句では複数の変数を同時に宣言することができます。

```
from Function func1, Function func2, Function func3
```

例として、`Function` 型の値を 3 つ宣言しました。このように書くと、`func1`, `func2`, `func3` それぞれにすべての関数が入ります。

つまり、

$$(\text{func1}, \text{func2}, \text{func3}) \in \text{Function} \times \text{Function} \times \text{Function}$$

みたいなことになります。使われていない変数や、無駄な計算のいくつかは最適化によって削られますが、いたずらに変数を増やすと思わぬ直積を生む原因になるので注意してください。

5.2 `where` 句

次に、`where` 句を用いて具体的に値をフィルタする方法を説明します。

`where` では、論理式を `and` や `or` を使ってつなげ、複雑な条件を記述することができます。

論理式には、`=`, `!=`といった演算や、predicate を用いることができます。

冒頭の `where` 句をもう一度考えてみましょう。

```
where fc.getTarget().getName() = "printf"
```

`fc` は `FunctionCall` の実体でした。すると、`getTarget` は `FunctionCall` に生えているメソッドのように見えます。実際にドキュメントを当たってみましょう。

「CodeQL C++ FunctionCall」等で調べるとドキュメントが出てきます。CodeQL は C++以外にもたくさんのライブラリを公開しているので、参照しているドキュメントが適切な言語のものか確認しましょう。

https://codeql.github.com/codeql-standard-libraries/cpp/semmle/code/cpp/exprs/Call.qll/type.Call$FunctionCall.html

少し下の方にスクロールすると、“Gets the function called by this call.” という predicate を見つけることができます。クリックして詳細を閲覧すると、次のようなシグネチャが確認できます。

```
1 Function getTarget()
```

`getTarget` は `Function` という実体を返すようです。`Function` は先程扱いましたね。

つまり、`fc.getTarget()`は、「`fc` が実際に呼んでいる関数の実体を手に入れる」という意味の式として解釈することができます。

続けて、`getName` も調べましょう。先程のページの帰り値の型の `Function` の部分をクリックすると `Function` のドキュメントを閲覧することができます。

`getName` の説明に、“Gets the name of this declaration.” とあります。また、同じようにシグニチャを確認すると、`string getName()` とあります。

ここに来てやっと身近な型が出てきました。`string` は皆さんの想像する文字列型だと考えていただいて良いです。どうやら、`getName` メソッドは `Function` の名前を文字列として返す predicate のようです。

これらを踏まえ、もう一度 `where` 句の条件を考察しましょう。

```
1 where fc.getTarget().getName() = "printf"
```

この `where` 句がどのような機能を持つのかなんとなく想像できるようになったと思います。つまり、「関数コール `fc` の内、`fc` が呼んでいる関数の名前が `"printf"`と一致するもの」を条件にしているのです。

我々が書いた `printf.c` の `main` では

```
1 printf("it works!");
```

とあり、この条件に合致した関数コールのはずです。`fc` には少なくともこの関数呼び出しが代入されるはずです（他の部分で `printf` を呼んでいる関数があった場合、それも条件に合致する）。

これで条件の絞り込みが終わりました。最後にこれを表示しましょう！あと少しです！

5.3 `select` 句

`from` で定義し、`where` で絞り込んだ値を、実際にどのように表示するのかを指定する句です。

```
1 select fc, fc.getEnclosingFunction(), fc.getLocation()
```

ここで、fc, fc.getEnclosingFunction(), fc.getLocation() の３つを表示しています。ここで使える predicate も先ほど参照したドキュメントにある通りです。

getEnclosingFunction は、その関数コールが行われた関数を、getLocation はその関数コールが行われた場所を返します。 printf("it works!"); は、main 関数内の関数呼び出しであり、コードとしては 4 行目に存在します。

```
|       fc       | col1 |                              col2                              |
+----------------+------+----------------------------------------------------------------+
| call to printf | main | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:4:5:4:10 |
```

クエリの実行によって得られた結果は我々の期待するものと一致していそうです。

5.4 まとめ

このように、ソースコードをデータベースライクに扱い、クエリを書くことによって情報を抽出することができました。

where では、指定した型の実体すべてを含む集合を計算し、where で型に付随している predicate を用いて条件の絞り込みを行い、最後に select によって値を表示する、といった感じです。

6 クエリの応用

更に複雑なクエリを書くため、もう少し応用的な文法を学びましょう。ですが、printf.c ではコードが小さく面白くないため、もう少し大きなコードベースを対象に解析を行います。

対象は何でも良いです。自分の興味のあるコードベースを選び、ビルドする時に CodeQL のデータベースを作成するようにしてください。

ここでは Linux Kernel の commit_creds 関数を対象に解析を行います。この関数に対する知識は必須ではありませんが、正確性を無視しざっくりと説明をすると「現在実行中のタスクの権限を更新する」という機能を持ちます。主に CTF 等で権限昇格に使われる馴染み深い関数であるため解析対象にしましたが、深い意味はありません。

カーネルのビルド方法に関してはここでは詳細に説明しません。いくつかの注意点として、デバッグ用の関数や KASAN などを有効にしている場合、抽出されるデータベースにも含まれるため、それらがノイズになる方は適宜 config を編集すると良いです。

また先程も少し言及しましたが、GCC でビルドを行うと一部ファイルが認識されなくなるバグが存在します。これは記事作成時の最新バージョン(2.21.3)でも再現します。少し試してみるだけであれば何でも良いですが、本格的に使いたい方は clang-17 を入れたうえで次のコマンドを実行してください。

```
codeql database create \
  --language=cpp \
  --command="make -j4 LLVM=1 LLVM_IAS=1" \
  kerneldb
```

それ以外は、先ほどの環境構築と同じことをすればオーケーです。

さて、作成したデータベースに対して早速クエリを書いていきましょう。

まず、先程学習したことを使って、「commit_creds が呼び出している関数一覧」を取得してみます。

```
1 import cpp
2
3 from Function commit_creds, FunctionCall callee_call, Function callee
4 where
5   commit_creds.getName() = "commit_creds" and
6   callee_call.getEnclosingFunction() = commit_creds and
7   callee_call.getTarget() = callee
8 select callee, callee.getLocation()
```

最初に、`commit_creds` という関数を探し、`FunctionCall` の中で `commit_creds` に含まれているものを取得します。最後に、その実体を `callee` として記録するという流れです。

```
1 |          callee          |                                col1
  |
2 +--------------------------
  +-------------------------------------------------------------------------+
3 | __builtin_constant_p     | file://:0:0:0:0
  |
4 | __builtin_expect         | file://:0:0:0:0
  |
5 | atomic_long_read         | file:///linux/include/linux/atomic/atomic-
  instrumented.h:3186:1:3186:16 |
6 | get_current              | file:///linux/arch/x86/include/asm/current.h:44:44:44:54
  |
7 （長いので省略）
```

おそらく大量の関数がリストアップされると思います。一つ一つクエリを改善していきましょう。

まず、`from` にて定義している `callee_call` ですが、これは最終結果には現れない、いわば「一時変数」のようなものです。この定義の規模なら問題になりませんが、より大きなクエリを書く準備のため、`from` を整理することから始めましょう。

CodeQL には、`from` 以外にも変数を宣言できる場所があります。その代表例が `exists` をはじめとする、量化子で変数を束縛する文法です。

これを使うと、先程のクエリを次のように書き換えることができます。

```
1  import cpp
2
3  from Function commit_creds, Function callee
4  where
5    commit_creds.getName() = "commit_creds" and
6    exists(FunctionCall fc |
7      fc.getEnclosingFunction() = commit_creds and
8      callee = fc.getTarget()
9    )
10 select callee, callee.getLocation()
```

exists の中で一時的に FunctionCall 型の fc を宣言しています。数学的な言い回しをするなら、「そのような fc が存在する」的な感じです。どのような fc か、それを記述しているのが | で区切られたあとの式です。内容は一つ前のものと変わりませんね。

これによって from による定義を一つ減らすことができました。exists 以外にも、forall や sum, concat など、表現力を上げるための文法や便利機能が沢山あります。ただ、名前から機能を容易に想像できる上、公式ドキュメントも充実しています。これらを使う頃には相応に理解度が深まっていると思うので、今回の記事では省略させていただきます。

次に、クエリの結果を見てみましょう。大量に出力されたエントリは、そのほとんどがコンパイラビルトインです。呼び出されている関数であることに変わりはないですが、解析対象としては不適です。これを弾く条件を書きましょう。コンパイラビルトインはソースコード中に定義を持たないという性質を利用し、論理式を構築します。

```
1  import cpp
2
3  from Function commit_creds, Function callee
4  where
5    commit_creds.getName() = "commit_creds" and
6    exists(FunctionCall fc |
7      fc.getEnclosingFunction() = commit_creds and
8      callee = fc.getTarget()
9    ) and
10 +  callee.hasDefinition()
11 select callee, callee.getLocation()
```

callee.hasDefinition() という部分を追加しました。これにより、定義を持たない関数群をまとめて除外することができます。このクエリを実行すると、ちょうど 10 個程度の結果が帰ってきます。余力がある方は、kernel/fork.c に存在する実装と見比べてみて、ちゃんと漏れなくリストアップされているか確認してみてください。

さて、ここから更に次のステップに進みたいところですが、その前段階として今作った「ある関数から呼ばれている関数一覧を取る」という操作を切り出して、再利用可能な形にしたいと思います。普通のプログラミングでいうところの、関数呼び出しみたいなものです。

先程のクエリを次のように書き換えて見ましょう。

```
1  import cpp
2
3  predicate calls(Function a, Function b) {
4    exists(FunctionCall fc |
5      fc.getEnclosingFunction() = a and
6      fc.getTarget() = b and
7      b.hasDefinition()
8    )
9  }
10
11 from Function commit_creds, Function callee
12 where
13   commit_creds.getName() = "commit_creds" and
14   calls(commit_creds, callee)
15 select callee, callee.getLocation()
```

predicate という単語が出てきました。これによって先程のロジックを calls として切り出しています。注意しなければならないのは、calls はあくまでも predicate、つまり述語であるという点です。

これも正確性を多少犠牲にした表現になりますが、最初は bool を返す関数のようなものだと考えると納得できます。where 句のフィルタは、各 predicate がすべて真のときに全体を真にするという認識で十分です。

from 句で Function を２つ定義しているため、カーネル内のすべての関数のペアが commit_creds と callee に入ります。まず commit_creds のほうは名前によってフィルタされ、次に callee の方は calls によってフィルタされます。「次に」という日本語を使いましたが、これらは逐次的に処理されているわけではなく、これらの条件を入れ替えても意味は変わりません。

```
1 from Function commit_creds, Function callee
2 where
3   calls(commit_creds, callee) and
4   commit_creds.getName() = "commit_creds"
5 select callee, callee.getLocation()
```

愚直に解釈すると、まずすべての caller callee ペアを見つけ、**その後**に commit_creds をフィルタしているように見えますが、それでは現実的な時間で応答するのは難しいでしょう。これを可能にしているのは、単純に最適化の精度が高いからです。クエリをどの順番で書いてもその性能が変わることは（おそらく）ありません。安心してわかりやすい順番で書けばよいです。

では、気を取り直してクエリの実装を続けましょう。次の目標は先ほどのクエリを応用し、「commit_creds から到達できる関数一覧」とします。つまりさっきのクエリを再帰的に適用してみよう！という感じです。

ちょうど、ここまで学習した文法と機能だけで、そのような機能を持つクエリを書けるような構成にしました。挑戦してみたい方は挑戦してみてください！

筆者の答えを次に示します。

```
predicate calls(Function a, Function b) {
  exists(FunctionCall fc |
    fc.getEnclosingFunction() = a and
    fc.getTarget() = b and
    b.hasDefinition()
  )
}

predicate calls_rec(Function a, Function b) {
  calls(a, b)
  or
  exists(Function transit | calls_rec(a, transit) and calls(transit, b))
}

from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  calls_rec(commit_creds, callee)
select callee, callee.getLocation()
```

`transit` という経由地点が**存在する**(`exists`)なら、そこから更に `b` に到達できるか、みたいな感じです。

実行すると、おおよそ 8000 エントリほどの結果が返ってくると思います。結果を見てみると、一見全然関係ない関数が含まれていますが、カーネルは複雑なので、どこかに `panic` でもあったのでしょう。そこから広がっているのだと思います。コンパイラビルトインを除外したときのように関係ないファイルや関数を一つ一つ取り除いても良いですし、これらをフィルタしていく作業は実際の解析において非常に重要なことですが、この記事の趣旨とは外れるためこれも割愛させていただきます。

その代わりに、ここでは賢く `maxdepth` を設定しようと思います。幸いにして、文字列比較に `string` を使ったように、`int` というプリミティブな型があるので、これを使って実装します。しかし、先程も強調しましたが、`predicate` は関数ではありません。「引数として `int maxdepth` を渡す」という認識でいると正しく実装できなくなってしまいます。

先程の説明の繰り返しになってしまいますが、`predicate` はあくまでも、引数の型を見て取りうる値を全部代入してみて結果が真ならば `true` を返す、と認識するべきです（正確にはこれも間違っていますが、詳細は後述します）。

それを踏まえ、次のようなクエリを考えてみます。

```
predicate calls_rec(Function a, Function b,int depth) {
  (calls(a, b) and depth = 1)
  or
  (exists(Function transit | calls_rec(a, transit, depth - 1) and
  calls(transit, b)))
}
from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  calls_rec(commit_creds, callee, 2)
select callee, callee.getLocation()
```

「深さ 2 の関数一覧を取る」という意図でこのクエリを書きました。直接的な関係がある時の `depth` は `1` であり、それ以外の場合は `depth-1` で `transit` まで到達できるなら該当、という感じです。

しかし、このクエリが終了することはありません。`predicate` は全部代入して確かめるということを意識すると、`int` として存在するすべての値を入れているため、膨大な時間がかかってしまうためです（CodeQL の int は 32bit 整数なので、無限ではない）。

これを判別する目安としては、実行に一分以上の時間がかかるか否かをチェックすればよいです。一分で終わらない場合、基本的にクエリが正しくないと考えて良いと思います。

この問題を解決するにはどうすればよいでしょうか？ 方法の一つとしては、`int` をそのまま使わないという手があります。

CodeQL には型の継承のような機能が存在し、コンストラクタ等に自身に対する条件を書くことで、そのクラスが持ちうる値の可能性を制御することができます。

```
class DepthLimit2 extends int {
  DepthLimit2() { this = [0 .. 2] }
}

predicate calls_rec(Function a, Function b, DepthLimit2 depth) {
  calls(a, b) and depth = 1
  or
  exists(Function transit | calls_rec(a, transit, depth - 1) and
  calls(transit, b))
}

from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  calls_rec(commit_creds, callee, 2)
select callee, callee.getLocation()
```

このようにすることで、現実的な時間で depth を指定できるようになります。指定する深さを 2 としてハードコードしていますが、CodeQL に Generics のような機能がないため、不可避の制限だと考えてください。

また、_ を使うことで、距離 2 以下で到達できる関数一覧を得ることもできます。

```
1  calls_rec(commit_creds, callee, _)
```

_ は don't care を意味し、何でも良いみたいな意味を持ちます。型として指定している DepthLimit2 は、0,1,2 のどれかを取るため、2 以下で到達できる関数一覧を取ることができます。

最後に、ここで学んだことを利用して、ちょっとしたクエリを書く練習をしてみましょう。回答には、count という言語機能を使っても構いません。 count は、条件に合う要素を数え上げることができます。例えば、num_func = count(Function f | f.getDefinition()) とすると、num_func には、定義をもつ関数の数が入ることになります。

問題:次の仕様を満たすクエリを書いてください。「commit_creds から呼び出される関数の内、そこから深さ 2 以下で到達できる関数の数を集計するクエリ」

結果はおおよそ次のようになると思います。

```
1  |       callee       | num_child |
2  +--------------------+-----------+
3  | atomic_long_read   |         5 |
4  | get_current        |         0 |
5  | key_fsgid_changed  |         5 |
6  | key_fsuid_changed  |         5 |
7  | put_cred_many      |         9 |
8  | get_cred           |         2 |
9  | dec_rlimit_ucounts |         3 |
10 | inc_rlimit_ucounts |         4 |
11 | _printk            |         2 |
12 | gid_eq             |         1 |
13 | uid_eq             |         1 |
14 | set_dumpable       |         4 |
15 | proc_id_connector  |        29 |
16 | cred_cap_issubset  |         3 |
```

答え:

```
1  from Function commit_creds, Function callee, int num_child
2  where
3    commit_creds.getName() = "commit_creds" and
4    calls(commit_creds, callee) and
5    num_child = count(Function child | calls_rec(callee, child, _))
6  select callee, num_child
```

6.1 まとめ

これにて、クエリの応用は終了です。まだまだ紹介しきれていない機能ばかりなのですが、すべてを紹介することはできないため、ドキュメントを理解する上で重要な思想や考え方を中心に説明してきました。

これ以上高度なことをしようするなら必ずドキュメントを読むことになるでしょう。しかし、ここで学んだ考え方を応用すれば容易に理解できるはずです。みなさんが面白いクエリを書き、楽しい CodeQL ライフを送れることを期待しています！

7 最小不動点

おまけパートです。

最後に、今まで紹介してきたクエリが内部でどのように処理されているのか紹介します。

クエリの作成に役立つかもしれないし、役立たないかもしれません。興味がある方だけ読んでいただければ結構です。

さて、本文中にも登場した `predicate` に関して、もう少し掘り下げてみましょう。より単純な例を考えます。

```
1  predicate getANumber(int x) {
2    x = 0
3    or
4    x <= 100 and getANumber(x - 1)
5  }
6
7  from int v
8  where
9    getANumber(v)
10 select v
```

これを実行すると、`0` から `100` までの整数が列挙されると思います。`0` から `100` の整数を `getANumber` に渡した場合、中の論理式が真になる、と言い換えることもできます。

`getANumber` の内容をよく見てみましょう。まず、`getANumber(0)`は真になります。`getNumber(1)`はどうでしょうか？ `x=0` の方は満たしませんが、`x <= 100 and getANumber(x - 1)`は満たします。なぜなら、`getANumber(0)`はすでに集合に含まれることを確認したからです。

同様に `x=2` も条件を満たします。これを繰り返していくと `x=101` までこの操作を続けることができ、そのタイミングで条件を満たさなくなります。実際の結果から、CodeQL も同じように考えていることがわかります。

では、内部のクエリエンジンはどのようにしてこれを計算しているのでしょうか？そのキーになるのが、**最小不動点**です。

クエリエンジンはまず、`getANumber` を満たすことができる引数の集合を空集合として考えます。次に、引数の型である `int` の中から適当な値を代入してみて、それが条件を満たすかチェックします。条件を満たすなら、集合に加えます。次に、拡大した集合を参照しつつ、

もう一度適当な値を `getANumber` に与えてみて、それが条件を満たすかどうかチェックします。新しく条件に合致する値が見つかった場合、それを集合に加えます。

この操作を繰り返していき、どこかのタイミングで集合の成長が止まる瞬間が来たとします。そのタイミングの集合を「`getANumber(x)`が真になる集合」として扱うことで、クエリを処理しています。

これを踏まえると、クエリの応用セクションで、評価が終了しないクエリの原因を推測することができます。

```
1 predicate calls_rec(Function a, Function b,int depth) {
2   (calls(a, b) and depth = 1)
3   or
    (exists(Function transit | calls_rec(a, transit, depth - 1) and
4
    calls(transit, b)))
5 }
```

まず適当な `a`, `b`, `depth` を与えてみて、それが条件を満たすなら追加、という操作を繰り返していきます。先程の例では、クエリエンジンの最適化によって `x` として与える値を適切に選び、現実的な時間でクエリを処理できました。

しかし、どうやら `calls_rec` は**複雑すぎる**ようで、`int` としてあり得るすべての値を代入してみて集合の構成を試みます。その結果、全く終わる気配がないクエリが誕生した、というカラクリです。

この事実を念頭に置くと、生の `int` や `string` をそのまま扱うのは危険な気がしてきます。この直感は正しく、`DepthLimit2` という制限した型を定義したように、クエリエンジンが無駄な探索を行わないように工夫するのが大切です。見識が広い方は、解集合プログラミングとのつながりに気がついたかもしれません。

ここで、ちょっと意地悪なクエリを考えてみましょう。先程のアルゴリズムによる集合の構成では、**集合の要素が減る**ような場合は考慮されていませんでした。そもそもそのようなクエリを書くことは可能でしょうか？

実は工夫をするとそんなクエリを書くことができます。考えてみたい読者の方は、ここで止めて考えてみてください。同様に、ここまで学んだ機能で書くことができます。

答え:

```
1 predicate p(int x) {
2   x = count(int e | p(e))
3 }
```

まず最初に、`0` を与えてみます。最初 `p` を満たす集合は空集合なので、`0` は条件に合致します。すると集合の個数が一つ増え、`count(int e | p(e))` は `1` になります。すると今度は `0` が条件を満たさなくなってしまいます。

このクエリを実際に処理するとどのような結果が得られるでしょうか？

CodeQL は賢く、このような意地悪なクエリの処理を拒否してきます。

```
1 ERROR: Non-monotonic recursion: target -!-> target (no_poset.ql:6,21-27)
```

つまり、内部でクエリを処理する際に、集合が縮まないことを要求しているのです。（詳しい人に指摘されるのが怖いので及び腰になりますが）数学的に表現するなら、集合間に包含関係によって半順序性を入れ、その順序に基づいた単調性を要求していると言い換えることもできます。

束論によると、半順序性を持った集合は、必ず最小不動点を持ちます（詳しくは、Knaster–Tarski の定理）。CodeQL においては、処理をする前段階で単調性を確かめることで、その後の最小不動点計算の結果が存在することを使っているようです。もちろん、それが現実的な時間で終わるかは判別してくれませんが。

これらは、ドメイン理論などとも関わりがありますが、筆者はそもそも数学に明るくないため、これ以上の言及は避けます。怖いので。

8 おわりに

ここで紹介した機能以外にも、たくさんの便利機能があり、マスターすることで日々のソースコードリーティングがより捗ることでしょう！

特に、CodeQL の目玉機能である、`taint tracking` はぜひ皆さんに紹介したかったのですが、内容が難解になりすぎてしまったので割愛しました。無念。

質問や誤植等がありましたら、気軽に連絡していただけると嬉しいです。

最後まで読んでいただきありがとうございました！

ゼロコスト AI コーディング

文　編集部　Till0196

1　Claude Code を Copilot Chat から使う

1.1　GitHub Pro の特典の Copilot Pro を活用しよう

GitHub Education の申請を通過すると、GitHub Pro 相当の特典が無償で付与されます。主な特典内容は、Copilot Pro やプライベートリポジトリの Actions Runner の稼働時間リミットなどがあります。この特典の目玉の一つが「Copilot Pro」です。Copilot のコード補完機能は有名ですが、実は「Copilot Chat」という対話型の AI アシスタントも利用できることをご存知でしょうか。

Copilot Pro で Copilot Chat として利用できるモデルは非常に多彩で、2025 年 6 月現在、以下のようなモデルがラインナップされています[†1]。

- GPT-4o
- GPT-4.1
- Claude Sonnet 3.5
- Claude Sonnet 3.7
- Claude Sonnet 3.7 Thinking
- Claude Sonnet 4
- Gemini 2.0 Flash
- Gemini 2.5 Pro
- o1
- o3-mini
- o4-mini

※Opus 4 や o3 などは Copilot Pro+プランでないと利用できないようです。

VSCode v1.99 以降に搭載された Copilot Chat の Agent モードは、単純な対話だけでなく、ワークスペース内のファイル群を横断して動作できる強力な機能です。しかし、実際に使ってみると、作業中に思考が終わらなくなったり、プロジェクトのディレクトリ構成をうまく把握してくれなかったりする傾向が見られます。

そこで気になってくるのが、賢いエージェントコーディングができると話題の「Claude Code」です。「Claude Code を利用するには、別途 Anthropic の API 契約や有料プランが必要

†1　GitHub Copilot のプラン - GitHub Docs
https://docs.github.com/ja/copilot/about-github-copilot/plans-for-github-copilot#%E3%83%A2%E3%83%87%E3%83%AB

では？」と考える方もいるでしょう。しかし、Copilot Pro ユーザーであれば、Claude Code を VSCode の Copilot から呼び出すことで、追加料金なしで利用することができます。

ただし、この方法には注意点もあります。あくまで Copilot Chat の言語モデルを利用するため、本来の Claude Code と比較して性能や精度が低下する場合があります。

とはいえ、VSCode 標準の Copilot Chat や Cursor の Agent Chat よりも、ワークスペースの把握能力やコンテキスト理解の面で優れており、タスク分解能力や状況把握に長けているため、かなり賢くコーディングタスクなどをこなしてくれます。

1.2 Claude Code を VSCode から呼び出してみよう

この記事では学生なら無料で使える Copilot Chat をすぐに利用できるため、VSCode を使用していますが、Anthropic の言語モデルが利用できる環境であれば、stdio の MCP サーバーに対応している AI エディタ（Cursor など）や AI ツール（Claude Desktop など）でも同様に実現することができます。

Claude Code のインストール

まずは、Claude の公式ドキュメント[†2] を参照しつつ、Claude Code をインストールします。

npm の環境があれば下記のコマンドでインストールできます。

```
npm install -g @anthropic-ai/claude-code
```

インストールが終わったら `claude` とコマンドを打つと「Welcome to Claude Code」などの文字列が表示されていれば準備は完了です。今回は `claude` コマンドで対話しないので、API の設定やログインも不要です。Ctrl + C などで CLI から抜けることができます。

VSCode で MCP サーバーの設定

続いて、具体的に VSCode から Claude Code を呼び出すための設定を行いましょう。

VSCode の設定ファイル(settings.json) に、MCP サーバーとしての設定を記述します。

1. VSCode でコマンドパレットを開きます。（Ctrl+Shift+P または Cmd+Shift+P）
2. `Open User Settings (JSON)` と入力し、settings.json ファイルを開きます。
3. 以下の JSON ブロックを追記します。

```
// settings.json
{
    "mcp": {
        "servers": {
            "claude_code": {
                "type": "stdio",
                "command": "claude",
                "args": ["mcp", "serve"]
            }
        }
    }
```

†2 Claude Code のセットアップ - Anthropic
https://docs.anthropic.com/ja/docs/claude-code/setup

```
12 }
```

設定の解説

この設定が何を行っているかを解説します。

- `"mcp"`: MCP(Model Context Protocol)の設定項目です。
- `"servers"`:複数の言語サーバーを定義するためのセクションです。
- `"claude_code"`:このサーバー設定の名前です。
- `“type”: VSCode とサーバーの通信方法を標準入出力（stdio）に指定しています。
- `"command": "claude"`:サーバーを起動するためのコマンドです。
- `"args": ["mcp", "serve"]`: `claude` コマンドに渡す引数です。これにより、CLI が LSP サーバーモードで起動します。

動作確認

1. `settings.json` を保存した後、VSCode を再起動します。
2. VSCode のチャット欄を開き、Agent モードを選択し、モデルに「Claude Sonnet 4」などの Claude 系モデルを設定します。
3.「`claude_code` を使って、現在のワークスペースの README.md を作成」などと送信すると、Claude Code が呼び出されるはずです。

図 1: 実際の動作イメージ

2 結びに

MCP サーバーを介して Claude Code を呼び出すことで、現状の Copilot Chat や Cursor の Agent Chat よりも優れたワークスペース理解とコンテキスト把握が可能になります。

近年の AI エージェントコーディング技術の進化は目覚ましく、特に Claude Code のようなツールは、単純なコード補完を超えて、プロジェクト全体の構造を理解し、複雑なタスクを適切に分解して実行することが可能となりました。

ただし、この手法で利用する Claude Code は、Copilot Chat の言語モデルを経由するため、呼び出し側のモデルに依存しており、本来の Claude Code API と比較して一部制限があります。それでも、日常的なコーディング作業やプロジェクト管理においてかなりの性能を発揮してくれます。

AI エージェントコーディングを効果的に活用することで、より創造的で本質的な開発作業に集中できます。この記事の内容を参考に、ぜひ皆さんも Claude Code を試してみて、従来

の LLM のチャット欄にコンテキストの説明とソースコードをコピペすることから解放され、現代的な開発体験を手に入れてみてはいかがでしょうか。

わ〜ど 読者アンケーど！

文　編集部　間瀬 BB

こんにちは、間瀬 BB(@bb_mase2)です。風邪が流行ったりなんなりしている今日この頃ですが、そんなことはなんのその、

暑い！

ので、勝手にアンケートを開催したいと思います（？）[†1] お題は以下の通りです！

エアコンの設定温度は何度？

- ☐ **2147483647 ℃**
- ☐ **28 ℃〜26 ℃**
- ☐ **25 ℃〜23 ℃**
- ☐ **22℃以下**
- ☐ **その他（自由記述欄へ）**

所属(任意) :

†1　記事がかけなかったなんて言えない

自由記述欄

記入が終わったら、破って 3A 棟、コナクリ前の WORD 置き場↓の今にも崩壊しそうなボックスに突っ込んでおいてください。回収して、次号で結果発表します。

編集後記

文　編集部　n4mlz

WORD 第 57 号「WORD 編集部も警備員とゲートを導入します号」をお読み頂きありがとうございます。編集長の n4mlz です。

さて、WORD 編集部は今日も部屋でいつも通り自由奔放にやっております。最近は部内で DJ がブームになっており、賑やかで大変楽しい毎日です。

もうそろそろ梅雨が明ける時期でしょうか。気象庁の発表によると、今年の梅雨入りは例年より遅く、また、梅雨明けもかなり早いようです。[†1] 九州〜近畿地方までは、既に梅雨明けを迎えているようですね。

しかし、「梅雨明け」とは一体何なんでしょう？実は、物理的に明確な判定があるわけではありません。これらは、地方ごとの気象台が独自に判断して発表するものです。気象台は毎日天気を確認して、「梅雨明け」を発表するかどうか考えているわけですね。また、気象台の担当する地方ごとに決められるため、「○○県の梅雨が明ける」という言い方はしません。梅雨前線の影響が明瞭でない北海道（冷帯）は、梅雨明けの発表対象外になっています。

また、気象台の「梅雨明け」の発表はやっていることは実質的に天気予報と同じなので、あくまで速報値です。秋頃になって、「やっぱあの時の梅雨明けは○○日だったな」と見直されることもあります（これを確定値と言います)。2022 年は 20 日以上修正された地域もあったようですね。ちなみに、あまりに梅雨が長引くと 8 月以降の秋雨と区別できなくなることから、立秋（8 月 7 日ごろ）までに梅雨明けを判断できない場合は、「梅雨明けを特定しない」として記録されるようです。

これから本格的な暑さが続きそうですが、ぼちぼち適度に気合を入れてやっていきましょう。熱中症対策も忘れずに。

†1 “https://www.data.jma.go.jp/cpd/baiu/sokuhou_baiu.html”

情報科学類誌

WORD

From College of Information Science

WORD 編集部も警備員とゲートを導入します号

発行者　情報科学類長

編集長　川崎晃太朗
筑波大学情報学群
情報科学類 WORD 編集部

制作・編集　（第三エリア C 棟 212 号室）

2025 年 7 月 9 日　初版第 1 刷発行　(128 部)