YAML で記述された設定ファイルの静的検査器の開発

中丸 智貴　千葉 滋

YAML は構造化されたデータを文字列として表現する方法の一つであり, その読み書きのしやすさ故に多くのソフトウェアで利用設定の記述法として採用されている. 当然ながら, それらソフトウェアの設定記述は YAML 記述であれば何でも良いというわけではない. ソフトウェアごとに “正しい” YAML の定義があり, あるソフトウェアの利用設定として正しい YAML 記述も, 別のソフトウェアの利用設定としては誤った YAML 記述になる. どのような YAML 記述が正しいかはドキュメント参照や試験実行によって調べることができる. しかし, そのような方法で “正解” を探すのはソフトウェアの利用者にとって大きな負担である. 利用設定の記述の静的検査器があれば, 利用者の負担を大幅に軽減することができる. そこで本研究では, YAML 形式の設定ファイルの静的検査器の開発に取り組んだ.

1 はじめに

継続的インテグレーションのための CircleCI[†1], コードレビューのための Sider[†2], 構成管理のための Ansible[†3], 静的 Web サイト生成のための Hugo[†4] など, 利用設定を YAML[†5] 記述によって行うソフトウェアは数多く存在する. XML や JSON など, 構造化されたデータを文字列として表現する方法は YAML 以外にもあるものの, 人間が読み書きがしやすいように設計されている YAML は比較的人気な利用設定の記述法である.

当然ながら, それらソフトウェアの設定記述は YAML 記述であれば何でも良いというわけではない. ソフトウェアごとに “正しい” YAML の定義があり, あるソフトウェアの利用設定として正しい YAML 記述も, 別のソフトウェアの利用設定としては誤った YAML 記述になる. 図 1 は CircleCI で Gradle[†6] プロジェクトをテストする設定例であり, 図 2 は Sider で PHP CodeSniffer[†7] を利用した検査を行う設定例である. どちらも YAML 記述としては妥当だが, 図 1 は Sider の設定としては不正であり, 逆に図 2 は CircleCI の設定としては不正である.

```
version: 2
jobs:
  build:
    docker:
      - image: circleci/openjdk:11-jdk
    steps:
      - checkout
      - run: ./gradlew check
```

図 1　CircleCI 設定例

```
linter:
  code_sniffer:
    options:
      standard: phpcs.xml.dist
```

図 2　Sider 設定例

どのような YAML 記述が正しいかはドキュメント

Tomoki Nakamaru, The University of Tokyo.
Shigeru Chiba, The University of Tokyo.

†1　https://circleci.com
†2　https://sider.review
†3　https://www.ansible.com
†4　https://gohugo.io
†5　https://yaml.org
†6　https://gradle.org
†7　https://github.com/squizlabs/PHP_CodeSniffer

参照や試験実行によって調べることができる．しかし，そのような方法で “正解” を探すのはソフトウェアの利用者にとって大きな負担である．ソフトウェアが高品質であるほど設定可能な項目は多く複雑で，正しい記述方法を見つけるためにドキュメント/QA サイト廻り，試験実行の繰り返しを多くの時間を奪われることもしばしばである．その中でも特に試験実行の繰り返しはコストが大きい作業である．SaaS (Software as a Service) やコンパイラの設定では，一度の試験実行で数十秒から数分という時間がかかることもあるためである．

利用設定の記述を静的検査器があれば，上述したような利用者の負担を大幅に軽減することができる．検査器と統合開発環境を連携させれば YAML 記述の編集中に単語補完や候補表示を行えるため，ドキュメント/QA サイトを巡回する頻度を減らすことができる．静的に (実行前に) 記述の誤りを発見できれば，試験実行に時間を割く必要もなくなる．

そこで本研究では，YAML 形式の設定ファイルの静的検査を行うための技術開発に取り組んだ．具体的には許容される YAML 記述を定義するための領域特化言語 (Domain-specific language, DSL)，そして定義された通りに YAML 記述が行われているかを検査するアルゴリズムの開発を行った．

以下ではまず開発した DSL と検査アルゴリズムについて説明し (第 2 節)，次に既存ツールや関連研究と合わせて今後の課題について説明する (第 3 節)．そして最後に結論を述べる (第 4 節)．

2 提案手法

提案手法を紹介するための例として，図 3 のような YAML 形式の文献情報記述を考える．文献情報を構成する項目は BibTeX を参考にしている．ここでは簡単のため，取り扱う文献の種類は会議録 (`inproceedings`) のみとする．

例の文献情報における “正しい YAML 記述” を，本研究で提案する DSL で定義したものが図 4 である．1 行目から 16 行目は各項目に記述できる値に関するルールである．18 行目以降は複数項目を跨ぐルールである．図 5 は提案する DSL の文法である．

```
author: Tomoki Nakamaru
title: The Best Programming Language
booktitle:
  Proceedings of the 1st Fake Conference
  on Programming Language
year: 2020
month: 9
pages: [1_1, 1_9]
address: 1-1-1, Bunkyo, Tokyo, Japan
organization: Fake press
```

図 3　YAML 形式での文献情報記述

```
1 {
2   author: string | [string],
3   title: string,
4   booktitle: string,
5   year: number,
6   editor: string | [string] | void,
7   volume: string | void,
8   number: string | void,
9   series: string | void,
10  pages: (string, string) | void,
11  address: string | void,
12  month: number | void,
13  organization: string | void,
14  publisher: string | void
15  *: void
16 }
17
18 { number: string } -> { volume: void }
19 { volume: string } -> { number: void }
```

図 4　文献情報 YAML の規則

図 6 は DSL で定義した通りに YAML の記述がされているか検査する関数 f の定義である．f は値とルールを取り，真偽値を返す関数である．b, n, s はそれぞれ YAML の真偽値，数値，文字列値を表し，e, v はどちらも任意の YAML データを表す．T, S, U は直和規則 (図 5 の ⟨unionRule⟩ に対応)，[...] は条件付き規則 (図 5 の ⟨conditionalRules⟩ に対応)，⟨...⟩ はルール全体 (図 5 の ⟨start⟩ に対応) を表している．

関数 f で行われる検査は直感的には以下のように説明できる: 検査は上位構造から下位構造へと進めていく．例えばある YAML データが ⟨listRule⟩ `[T]` に従っているか検査する場合，まずデータがリスト型で

$$
\begin{array}{rcl}
\langle \text{start} \rangle & ::= & \langle \text{unionRule} \rangle\ \langle \text{conditionalRules} \rangle \\
\langle \text{unionRule} \rangle & ::= & \langle \text{singleRule} \rangle \mid \langle \text{singleRule} \rangle\ \texttt{|}\ \langle \text{unionRule} \rangle \\
\langle \text{singleRule} \rangle & ::= & \langle \text{valueRule} \rangle \mid \langle \text{containerRule} \rangle \\
\langle \text{valueRule} \rangle & ::= & \texttt{void} \mid \texttt{null} \mid \texttt{boolean} \mid \texttt{number} \mid \texttt{string} \\
\langle \text{containerRule} \rangle & ::= & \langle \text{listRule} \rangle \mid \langle \text{tupleRule} \rangle \mid \langle \text{mapRule} \rangle \\
\langle \text{listRule} \rangle & ::= & \texttt{[}\ \langle \text{unionRule} \rangle\ \texttt{]} \\
\langle \text{tupleRule} \rangle & ::= & \texttt{(}\ \langle \text{elementRules} \rangle\ \texttt{)} \\
\langle \text{mapRule} \rangle & ::= & \texttt{\{}\ \langle \text{keyRules} \rangle\ \texttt{\}} \\
\langle \text{elementRules} \rangle & ::= & \langle \text{nonEmptyElementRules} \rangle \mid \epsilon \\
\langle \text{nonEmptyElementRules} \rangle & ::= & \langle \text{elementRule} \rangle \mid \langle \text{elementRule} \rangle\ \texttt{,}\ \langle \text{nonEmptyElementRules} \rangle \\
\langle \text{keyRules} \rangle & ::= & \langle \text{nonEmptyKeyRules} \rangle \mid \epsilon \\
\langle \text{nonEmptyKeyRules} \rangle & ::= & \langle \text{keyRule} \rangle \mid \langle \text{wildcardRule} \rangle \mid \langle \text{keyRule} \rangle\ \texttt{,}\ \langle \text{nonEmptyKeyRules} \rangle \\
\langle \text{keyRule} \rangle & ::= & \langle \text{keyName} \rangle\ \texttt{:}\ \langle \text{unionRule} \rangle \\
\langle \text{wildcardRule} \rangle & ::= & \texttt{*}\ \texttt{:}\langle \text{unionRule} \rangle \\
\langle \text{keyName} \rangle & ::= & \textit{key name} \\
\langle \text{conditionalRules} \rangle & ::= & \langle \text{nonEmptyConditionalRules} \rangle \mid \epsilon \\
\langle \text{nonEmptyConditionalRules} \rangle & ::= & \langle \text{conditionalRule} \rangle \mid \langle \text{conditionalRule} \rangle\ \langle \text{nonEmptyConditionalRules} \rangle \\
\langle \text{conditionalRule} \rangle & ::= & \langle \text{unionRule} \rangle\ \texttt{->}\ \langle \text{unionRule} \rangle
\end{array}
$$

図 5　提案する DSL の文法

$$
\begin{array}{rcl}
f(\texttt{null}, \texttt{null}) & = & \mathit{true} \\
f(b, \texttt{boolean}) & = & \mathit{true} \\
f(n, \texttt{number}) & = & \mathit{true} \\
f(s, \texttt{string}) & = & \mathit{true} \\
f(\texttt{[]}, \texttt{[}T\texttt{]}) & = & \mathit{true} \\
f(\texttt{[}e\texttt{]}, \texttt{[}T\texttt{]}) & = & f(e, T) \\
f(\texttt{[}e_1\texttt{,}e_2\texttt{,}\ldots\texttt{]}, \texttt{[}T\texttt{]}) & = & f(e_1, T) \land f(\texttt{[}e_2, \ldots\texttt{]}, \texttt{[}T\texttt{]}) \\
f(\texttt{[]}, \texttt{()}) & = & \mathit{true} \\
f(\texttt{[}e\texttt{]}, \texttt{(}T\texttt{)}) & = & f(e, T) \\
f(\texttt{[}e_1\texttt{,}e_2\texttt{,}\ldots\texttt{]}, \texttt{(}T_1\texttt{,}T_2\texttt{,}\ldots\texttt{)}) & = & f(e_1, T_1) \land f(\texttt{[}e_2\texttt{,}\ldots\texttt{]}, \texttt{(}T_2\texttt{,}\ldots\texttt{)}) \\
f(\texttt{\{\}}, \texttt{\{\}}) & = & \mathit{true} \\
f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{\}}) & = & \mathit{true} \\
f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{}l\texttt{:}T\texttt{\}}) & = & (k_1 \in \{l\} \land f(v_1, T)) \lor \ldots \\
f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{}l_1\texttt{:}T_1\texttt{,}l_2\texttt{:}T_2\texttt{,}\ldots\texttt{\}}) & = & f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{}l_1\texttt{:}T_1\texttt{\}}) \land f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{}l_2\texttt{:}T_2\texttt{,}\ldots\texttt{\}}) \\
f(\texttt{\{\}}, \texttt{\{*:}T\texttt{\}}) & = & \mathit{true} \\
f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{*:}T\texttt{\}}) & = & f(v_1, T) \land \ldots \\
f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{}l_1\texttt{:}T_1\texttt{,}\ldots\texttt{,*:}S\texttt{\}}) & = & f(\texttt{\{}k_1\texttt{:}v_1\texttt{,}\ldots\texttt{\}}, \texttt{\{}l_1\texttt{:}T_1\texttt{,}\ldots\texttt{\}}) \land (k1 \in \{l_1, \ldots\} \lor f(v_1, S)) \land \ldots \\
f(o, [T\texttt{->}S]) & = & \neg f(o, T) \lor f(o, T) \land f(o, S) \\
f(o, [T_1\texttt{->}S_1, T_2\texttt{->}S_2, \ldots]) & = & f(o, [T_1\texttt{->}S_1]) \land f(o, [T_2\texttt{->}S_2, \ldots]) \\
f(o, \langle T, () \rangle) & = & f(o, T) \\
f(o, \langle T, [S_1\texttt{->}U_1, \ldots] \rangle) & = & f(o, T) \land f(o, [S_1\texttt{->}U_1, \ldots]) \\
f(\_, \_) & = & \mathit{false}
\end{array}
$$

図 6　検査関数 f の定義

あるか検査し, その後 (リスト型である) データの各要素が T に従っているかを調べる. 基本的にこれは再帰的に繰り返して真偽値へと還元することで, 与えられた YAML の正否判定を行う. 検査中, ⟨mapRule⟩ に書かれたキーの出現順序は考慮しない. また * は, それが記述されている mapRule の他のキーにマッチしない全てのキーに対するルールを指定している.

3 関連研究と今後の課題

第 2 節で紹介した提案手法により基本的な設定記述の検査は行えるようになった. しかし現実世界での応用を進めるには未だ多くの課題が存在する. 以下では関連研究と合わせ, 今後の課題をまとめる.

型の記述はできるようになったものの, 現実世界の設定ファイルに対して図 4 のような表記を書くのは非常に手間がかかる作業である. 文献 [3] と文献[1] で取り組まれているように, サンプルデータからの型推定が行えれば, 既存ソフトウェアにおける正しい YAML 記述の定義を容易に得ることができる. しかし, 文献 [3] と文献[1] で提案されている方法は 設定ファイルのルールという文脈では不十分な点が多い. YAML 設定での応用を見据えた型推定アルゴリズムの開発は, 今後の第一の課題である.

複数項目に跨がる条件 (図 4 の 18 行目以降) の定義は簡素なものであり, ある程度のルールを記述可能であるものの, 未だ十分ではない. XML の場合は, XDuce [2] などが存在しており, XDuce を用いれば様々な検査が行える. 本研究の特徴は YAML 形式の設定記述に特化し, よりルールを簡潔に記すことができるようにしている点である. しかし現状, YAML 設定記述でありがちな「ある map の中には $k_1, k_2, ...k_n$ を指定できるが, 同時に指定して良いのは 1 つだけ」というようなルールのために似たような条件付きルールを n 個記述する必要があるなど, あまり利便性が高いとは言えない状態である. 今後 DSL を拡張し, YAML の設定記述のルール記述の利便性を高めることは重要な課題である.

先述した通り, 現実世界のソフトウェアの設定ファイルのルールは非常に多くの項目を持つ. そのような状況ではルール数も多く, 条件を記述するうちに, 成り立ち得ないルールや任意の YAML が不正と判断されるルールを記述してしまいかねない. このようなルールの誤りを静的に検査する機能の開発も今後の課題である.

また, 既存ソフトウェアの利用者の支援だけでなく, 今後 YAML によって設定を行えるソフトウェアを作る開発者の支援も重要である. 例えば図 4 のようなルールから, Java のクラス定義を生成すれば, 開発者が利用者の記述した設定の検査を行う必要はほとんどなくなる. しかしその場合, 直和規則の存在は取り扱いが自明ではない. 検査する上では直和規則の項のどれに合致しているかは重要ではないが, 実際にインスタンスを作成する場合には具体的にどの項に合致しているか明らかでなければならない. (このような問題があるため, YAML validator[†8] では直和規則の記述はできない. しかし直和規則は YAML の設定記述のルール記述に必要不可欠なものである.) 問題を解決する方法を見つけ, 開発者支援に取り組むのも今後の課題の一つである.

今後の研究では上に挙げた課題に取り組み, YAML 形式の設定記述の静的検査を広く普及させることを目指す.

4 まとめ

本研究では YAML 形式の設定ファイルの静的検査を行うため, 許容される YAML 記述を定義する DSL, 定義された通りに YAML 記述が行われているかを検査するアルゴリズムの開発を行った. 提案した手法は基本的な検査が行えるのみではあるものの, 今後さらに研究を進め, 現実世界のソフトウェア開発現場に貢献することを目指す.

参 考 文 献

[1] Fisher, K., Walker, D., Zhu, K. Q., and White, P.: From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data, *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008, pp. 421–434.

[2] Hosoya, H. and Pierce, B. C.: XDuce: A Stat-

†8 https://www.npmjs.com/package/yaml-validator

ically Typed XML Processing Language, *ACM Trans. Internet Technol.*, Vol. 3, No. 2, pp. 117–148.

[3] Petricek, T., Guerra, G., and Syme, D.: Types from Data: Making Structured Data First-Class Citizens in F#, (2016), pp. 477–490.