構造的部分型付けと解析表現文法によるスキーマ言語

山口 真弥　倉光 君郎

スキーマによる検証は，Web 上の半構造化データの信頼性を高める重要な技術である．しかし，JSON データを扱う多くのアプリケーションではスキーマ検証はほとんど活用されていない．本論文では，よりアプリケーション開発者が利用しやすいスキーマとして，構造的部分型付けと解析表現文法 (PEG) による JSON 向けスキーマ言語 Celery を提案する．構造的部分型付けは，スクリプト言語など多くの動的言語で暗黙的に用いられており，JSON データの柔軟さと相性が良いと考えられる．PEG は、正規表現を上回る表現力があり，統合的な文字列パターンの検証が可能になる．また，PEG を用いる利点は，スキーマ構造を PEG に変換することができ、再帰降下構文解析法でスキーマ検証自体を行える．本論文では，Celery の基本的な特徴と，PEG によるスキーマ検証アルゴリズムについて述べる．

Schema validation is an integral part of reliable semi-structured data on the Web. However, the schema validation in many applications that handle JSON data is rarely taken advantage. In this paper, we present a JSON schema language:Celery that is based on Parsing Expression Grammar (PEG) and structural subtyping for developer's usability. Structural subtyping has been used implicitly in many dynamic languages, such as scripting language, so we consider that there is a compatibility with JSON. PEG has expressive power that beyond the regular expression and allows validation for integrated string patterns. It is also advantage of adapting PEG that we can convert schema structures to PEGs and validate by recursive descent parsing. In this paper, we present the principles of Celery's design and PEG-based schema validation algorithm.

1　はじめに

スキーマによる検証は，Web 上の半構造化データの信頼性を高める重要な技術である．Web アプリケーションはスキーマ検証によって，入力，あるいは出力データが期待される内容を持っていることを保証することができる．

Web アプリケーションでのデータ交換に広く用いられているフォーマットの一つとして JSON がある．JSON には標準化されたスキーマ言語が存在しないため，アプリケーションの開発者はスキーマ検証を活用しない場合が多い．しかし，JavaScript をはじめとした動的言語ではダックタイピングによってオブジェクトを参照するため，プロパティの存在や，データの型が保証されないという問題がある．

また，効率的なスキーマ検証器の実装が簡単でないことも，スキーマが活用されない一因と言える．一般的なスキーマ検証器は木オートマトン理論に基づいている．すなわち，スキーマ検証を実行する際，入力サイズに比例して木を構築するためのメモリを必要とするため，効率的な検証器の実装にはコストが掛かる．理論上ではストリーミングによる検証アルゴリズムは研究されているが [4] [6]，依然として既存の実装のほとんどは木オートマトンを利用しているため，データのサイズが大きくなるほど性能が劣化してしまうという欠点を持つ．

本論文では，構造的部分型付けと解析表現文法による JSON 向けスキーマ言語 Celery を提案する．構造的部分型付けはスクリプト言語などの多くの動的

A Schema Language Based on Structural Subtyping and PEG

Shinya Yamaguchi, 横浜国立大学, Yokohama National University.

Kimio Kuramitsu, 横浜国立大学, Yokohama National University.

型付け言語で暗黙的に用いられている型付けの形式であり，柔軟性のある JSON データとの相性が良いと考えられる．また，Celery のスキーマ検証は解析表現文法 (PEG) に基づいている．スキーマ制約を PEG に変換することによって，再帰降下構文解析法でスキーマ検証そのものを実行できる．本論文では，Celery の基本的な特徴と PEG によるスキーマ検証アルゴリズムについて述べる．

2 動機

スキーマの主な役割は，データベース設計の仕様を表すことである．Celery は，開発者にとっての扱いやすさを重視して設計されている．本節では，Celery の設計動機を，例を交えて説明する．

2.1 JSON と JavaScript

JSON(JavaScript Object Notation, [1]) は，Ecma International によって標準化されたデータフォーマットである．その名前が示す通り，JSON は，JavaScript のオブジェクトリテラルとして用いられている．また，形式がシンプルであり，可読性が高いため，多くのプログラミング言語やデータベースシステムにおいてデータ交換形式として採用されている．

以下に JSON オブジェクトの簡単な例を示す．

```
book = {
    "title": "Paper Towns",
    "year": 2009
}
```

{} はオブジェクトのコンストラクタであり，そのプロパティはコロン記号 (:) で区切られたキーと値の組み合わせによって表される．

JavaScript は動的型付け言語であるため，JSON オブジェクトには型が付けられていない．すなわち，オブジェクトの構造を定義するための記法が用意されていない．プログラマは `book` というオブジェクト中の `title` という値を，`book.title` や `book["title"]` と記述することで自由に参照することができる．*name* の型は実行時に動的に決定され，特定のプロパティがオブジェクト中に存在しなかった場合には，未定義の値として `undefined` が返される．動的な言語におけるこのような柔軟な性質は**ダックタイピング**と呼ばれている．ダックタイピングによって，プログラマは型定義に縛られないコーディングが許される．一方で，オブジェクトプロパティの存在そのものやデータの型は保証されない．これは，Web 上でのデータ交換を考える上で重大な問題になる．本論文では，柔軟性を保ちつつ，型安全にオブジェクトの交換が行われることを目指す．

2.2 構造的部分型付け

構造的部分型付けは，部分型の関係が構造によって決定される型付けの形式であり，OCaml や Scala のような強力な静的型付けプログラミング言語で採用されている．[9] また，多くの動的言語において暗黙的に用いられていることが知られている．Java や C# に採用されている**名前的部分型付け**とは異なる．

```
class T1 {x, y, z}
class T2 {x, y}
class T3 extends T2 {z}
```

例として，$T1, T2, T3$ という 3 つの異なる型が定義されている場合を考える．上記の例はそれらの型を Java に似た記法を用いて定義したものである．キーワード `extends` は，T3 が T2 の全てのプロパティを継承することを明示していることに注意されたい．名前的型付けと構造的型付けの差異は型の等価性と部分型の関係性にある．T1 と T3 は名前的型付けにおいて，型名が一致しないため，全く異なる型として扱われる．一方で，構造的型付けにおいては 2 つの型は全く同じプロパティを持つため，T1 と T3 は同じ型として扱われる．同様に，名前的型付けでは継承関係が明記されていないため T1 は T2 の部分型ではないが，構造的型付けでは T1 は T2 の部分型であるとみなされる．このように，構造的部分型付けは柔軟に型を判別することが可能である．特に半構造化データでは様々なデータを追記可能であるため，構造的部分型付けとの相性が良いと考えられる．

注意しなければならない重要な点は，同名のプロパティである T1 の x と，T3 の x がそれぞれ異なる

内容を持っていた場合である．構造的型付けではこれらの不一致を検知することができない．一方，名前的型付けでは，型名は名前空間の接頭語として作用するため，2 つの同名プロパティを区別することができる．この点で，名前的型付けは，より厳格なスキーマ検証を提供するといえる．しかしながら，この問題は他のプロパティが 2 つのスキーマ間で全て一致している場合でのみ発生する．現実にそのようなケースが発生することは稀であり，問題にはならない．

以上より，構造的部分型付けはスキーマ検証に適しており，十分に実用的な型付け形式であると考えられる．

3 スキーマ言語 Celery

Celery はデータ構造を表現するためのスキーマ言語である．本節では，Celery を形式的に定義し，例を挙げて Celery の基本的な特徴を説明する．

3.1 型定義

Celery で用いられる型 `T` は以下の形式で定義される．

`T`	:=	`X`	型名
		`boolean`	真偽値型
		`int`	整数型
		`float`	浮動小数点数型
		`string`	文字列型
		`any`	すべての JSON 形式
		`T[]`	配列型
		`enum { v* }`	列挙型

図 1 Celery 形式定義

ここで，型名 `X` は以下の形式で定義されたオブジェクト型の名前を表す．

```
struct X = ( n: T )*
```

3.2 構文

図 2 は，JSON データの一例であり，プロパティの一つ（`related`）がネストした構造を持っている．図 3 はこの例に対応する Celery のスキーマ定義を表している．

```
{
    "title": "Go Set a Watchman",
    "author" : "Harper Lee",
    "year" : 2009
    "rate"     : 4.53
    "related": [
     {
       "title": "All the Light",
       "author" : "Anthony Doerr"
     },
     {
       "title": "The Martian"
       "author": "Andy Weir"
     }
    ]
}
```

図 2 JSON の例

`struct` によって宣言されたオブジェクト型のプロパティは `[プロパティ名]:[型]` という形式で表され，順不同に扱われる．プロパティには主キー制約が存在し，同名同型の組み合わせが重複することは認められない．また，型宣言では，Book 型のプロパティ `related` のように，外部のオブジェクト型 `RelatedBook` を参照することが可能である．このようにオブジェクトごとにスキーマを定義することによって，型定義を繰り返し書く必要がなくなり，可読性が向上する．

Celery のスキーマ検証器は，構造的部分型付けによって検証を行う．図 3 のスキーマにおいて `rate` は未定義のプロパティであるが，図 2 のオブジェクトは Book 型の全ての要素を持っている．そのため，スキーマ検証器は図 2 のオブジェクトを Book 型の部分型として受理する．

4 解析表現文法

解析表現文法 (Parsing Expression Grammar, PEG, [3]) は，様々なプログラミング言語のコミュニティから注目されている，実用的な文法基盤である．PEG の表現力と構文解析のアルゴリズムは，データのスキーマ検証に適していると考えられる．本節では PEG の性質を端的に説明し，スキーマ検証に対する実用性について述べる．

```
struct Book =
            title:string
            author:string
            year:  int
            related:RelatedBook[]

struct RelatedBook =
            title:string
            author:string
```

図 3　Celery の例

表 1　PEG の演算子

PEG	Type	Description
' '	Primary	Matches text
[]	Primary	Matches character class
.	Primary	Any character
A	Primary	Non-terminal application
(p)	Primary	Grouping
$p?$	Unary suffix	Option
$p*$	Unary suffix	Zero-or-more repetitions
$p+$	Unary suffix	One-or-more repetitions
$\&p$	Unary suffix	And-predicate
$!p$	Unary suffix	Negation
$p_1 p_2$	Binary	Sequencing
p_1 / p_2	Binary	Prioritized Choice

4.1　文法，表現，演算子

解析表現文法 G は 4 つの要素からなる組 $G = (N, \Sigma, P, p_s)$ によって形式的に表される．N は非終端記号の有限集合，Σ は終端文字の有限集合，P は表現の有限集合，p_s は開始記号を表す．ここで，我々は，変数 a, b, c を終端記号，A, B, C を非終端記号，p を解析表現として扱う．各文法規則は，バッカスナウア記法に似た形式 $A = p$ で表される．これは非終端記号 A から解析表現 p へのマッピングである．

表 1 は，PEG の演算子とその説明をまとめたものである．文字列 $'abc'$ は同一の入力にマッチし，$[abc]$ は a, b, c のいずれか一つにマッチする．. 演算子は任意の 1 文字とマッチする．$p?$，$p*$，$p+$ 表現は，最長位置まで貪欲にマッチすることを除けば，一般的な正規表現と同様に振る舞う．p_1 p_2 は 2 つの表現 p_1 と p_2 が連接しているかどうか試し，もし失敗すれば開始位置までバックトラックする．p_1/p_2 は最初に p_1 を試し，もし p_1 が失敗した時にのみ p_2 を試す．表現 $\&p$ は p を試し，マッチしても文字列を消費しない．一方で表現 $!p$ は p とのマッチングが失敗した時に成功し，p が成功した時は失敗する．

例として JSON の構文に対応する PEG 文法を図 4 に示す．文法は読みやすさのため単純化されているが，14 の文法規則によって JSON の構文を定義することができており，シンプルかつ強力であると言える．PEG では字句解析を用いないため，文法には字句的な要素と構文的な要素の両方を統合して定義することに注意されたい．

4.2　構文解析アルゴリズム

スキーマ検証において，処理が高速であることと，実装しやすいことは重要な要素である．通常，PEG による構文解析器は，入力長に対する空間計算量が一定であるトップダウン型再帰下降構文解析アルゴリズムによって実装される．また，多くのプログラミング言語において，再帰下降構文解析器の実装は単純かつ簡単である．既存のパーサジェネレータの存在が，その実装のしやすさを裏付けている．

PEG の有名な欠点として，バックトラックによる最悪指数時間計算量が挙げられる．[2] しかしながら，JSON の構文規則は非常に限定されているため，1 シンボル先読みすることによってバックトラックを回避することが可能である．つまり，PEG による JSON データの構文解析は線形時間で行われると考えられる．加えて，PEG ではアルゴリズムレベルでの最適化が多く存在し，それらを適用することによって更なる高速化が期待できる．これらの PEG の性質が，スキーマ検証器の実装において実用性をもたらすものと考える．

5　スキーマ検証のための PEG 変換

Celery のスキーマ検証は，PEG ベースの再帰下降構文解析器によって行われる．すなわち，Celery で記述された構造的な制約を，JSON の構文を表す PEG へと変換して実行される．本節では Celery を

```
File       = S* Value S* !.
Value      = String / Number / Object / Array
           / Null / True / False
Object     = '\{' Member (',' Member )* '\}'
Member     = String ':' Value
Array      = '[' Value (',' Value )* ']'
String     = '"' (!'"' .)* '"' S*
True       = 'true' S*
False      = 'false S*
Null       = 'null' S*
Number     =  '-'? INT (FRAC EXP? / '') S*
INT        = '0' / [1-9] [0-9]*
FRAC       = '.' [0-9]+
EXP        = [Ee] ( '-' / '+')? [0-9]+
S          = [ \t\r\n]
```

図 4　JSON の構文を表す PEG の文法

P_{OBJECT} = '{' P_{MEMBER} (',' P_{MEMBER})* '}'
P_{MEMBER} = P_{STRING} ':' P_{ANY}
P_{ARRAY} = '[' P_{ANY} (',' P_{ANY})* ']'
P_{STRING} = '"' (!'"' .)* '"'
$P_{BOOLEAN}$ = 'true' / 'false'
P_{INT} = '0' / [1-9] [0-9]*
P_{FLOAT} = P_{INT} '.' [0-9]+
P_{ANY} = P_{OBJECT} / P_{ARRAY} / $P_{BOOLEAN}$ / P_{FLOAT} / P_{INT} / P_{STRING}

図 5　JSON 構文を表現する構文規則

PEG へと変換するアルゴリズムと，検証器の実装について説明する．

変換アルゴリズムの説明では，JSON におけるオブジェクトやプロパティの名前 n に対応する PEG の構文規則を P_n という形式で表す．また，JSON の構文を表すための基本的な規則を図 5 に予め定義しておく．これらの規則は Celery から PEG への変換において共通して用いられる．

5.1　変換アルゴリズム

まず，Celery で定義されたオブジェクト型の変換から説明していく．なお，変換後の構文規則は簡略化されていることに注意されたい．

struct で宣言されたオブジェクト型 X に対応する PEG の構文規則を P_X とすると，X は以下のように変換される．

- P_X = '{' (P_{x_1} / P_{x_2} / ... / P_{x_n} / P_{ANY})* '}'
- P_{x_n} = '"x_n"' ':' $\tau(x_n)$

構文規則 P_X 内の最後の選択肢である P_{ANY} は構造的部分型付けを許すための表現である．P_{x_n} は，オブジェクト型 X で定義されている n 番目のプロパティを表している．オブジェクト型のプロパティは順不同であるため，このような変換は正確ではない．加えて，純粋な PEG では，いわゆる主キー制約を表現することができないため，実際には PEG の拡張構文を利用する必要がある．ただし，基本的な変換の形式は変化しないため，本節ではこのまま議論を進める．拡張構文による実装については後述する．

構文規則 P_{x_n} 内で定義されている $\tau(x)$ は，オブジェクト型のプロパティ x を入力とし，対応する PEG を出力する変換関数である．図 1 で定義した Celery の型は以下のように変換される．

- $\tau(X) = P_X$
- $\tau(boolean) = P_{BOOLEAN}$
- $\tau(int) = P_{INT}$
- $\tau(float) = P_{FLOAT}$
- $\tau(string) = P_{STRING}$
- $\tau(any) = P_{ANY}$
- $\tau(T[]) =$ '[' $\tau(T)$ (',' $\tau(T)$)* ']'
- $\tau(enum\ \{v_1, ..., v_n\}) =$ '"v_1"' / ... /'"v_n"'

5.2　変換器の実装

前節で述べたように，純粋な PEG では順不同なシーケンスや主キー制約を直接表現することができない．それらのスキーマ制約を表現するためには，PEG 処理系に実装されているセマンティックアクションや拡張構文を用いる必要がある．したがって，変換器の実装に際して，強力な表現力を持つ PEG 処理系を選択することは重要である．

本論文では，PEG パーサライブラリの一つである Nez [7] を利用して Celery-to-PEG 変換器の実装を行う．Nez は文法変換器やパーサ生成器のためのフレームワークを提供しており，我々の変換器の実装もその一部として統合されている．つまり，Celery ファイルを Nez の文法定義として直接読み込み，Nez に備わっているインタプリタ型のパーサでスキーマ検証を

表 2　Nez 拡張構文

PEG	Description
`<uniq e>`	構文規則 e が 1 回のみ受理される
`<set x y e>`	構文規則 e で受理された文字列がシンボル x と y を含む

行うことができるため，個別のスキーマ検証器を用意する必要がないという利点がある．また，Nez の文法には，シンボルテーブルや AST の生成を可能にする PEG の拡張構文が存在するためスキーマ制約の変換に適している．

拡張構文を用いた変換の簡単な例として，オブジェクト型 X が int 型のプロパティ x_1 と string 型のプロパティ x_2 を持つ場合を考える．

$$\begin{aligned}\texttt{struct X} = {} & x_1 : \texttt{int} \\ & x_2 : \texttt{string}\end{aligned}$$

このとき，Nez の文法への変換は以下のように表される．

- $P_X = '\{' < \texttt{set}\ P_{x_1}\ P_{x_2}\ (\ P_{x_1}\ /\ P_{x_2}/\ P_{ANY}\ )* > \ '\}'$
- $P_{x_1} = < \texttt{uniq}\ '"x_1"'\ ':'\ P_{INT} >$
- $P_{x_2} = < \texttt{uniq}\ '"x_2"'\ ':'\ P_{STRING} >$

5.1 節で定義した変換との差異は，拡張構文である set，uniq が追加されたことである．今回の実装で用いる拡張構文とその意味を表 5.2 にまとめる．

構文規則 P_X における表現 set 演算子は，$(\ P_{x_1}\ /\ P_{x_2}\ /\ P_{ANY}\ )*$ によって受理される入力が，シンボル P_{x_1} と P_{x_2} を含んでいれば成功する．また，P_{x_1}，P_{x_2} では，uniq 演算子によってそれぞれが 1 回のみ受理される．結果として，この 2 つの拡張構文によって，順不同なシーケンスと主キー制約を PEG で表現することが可能になる．

6　関連研究

JSON と比較して，XML ではスキーマ検証や処理系の様々な研究が行われてきた．静的型付けプログラミング言語 XDuce[5] は型システムと木オートマトン理論[8] に基づいている実用的な XML 処理系の一つである．XDuce では，XML データに対応するスキーマを正規表現型という形式で表現する．しかし，XDuce は純粋な正規表現を用いているため，順不同なシーケンスや主キー制約をスキーマ自体に定義することはできない．その他にパターンマッチやタグの検査，部分木の抽出，動的なスキーマ検証など，XML を処理する様々な機能を提供している．

7　結論

本論文では，JSON 向けスキーマ言語 Celery の特徴と PEG ベースのスキーマ検証アルゴリズムについて述べた．Celery はユーザにとって分かりやすくシンプルな構文でスキーマを表現することができる．また，構造的部分型付けを採用することによって，JSON データを柔軟に検証する能力を持つ．さらに，Celery を拡張された PEG へと変換し，再帰下降構文解析器でスキーマ検証を実行することで，PEG の強力な解析能力を得るだけでなく，スキーマ検証器の実装コストを抑えられることを示した．

参 考 文 献

[1] ECMAInternational: The JSON Data Interchange Format. http://www.ecma-international.org/publications/standards/Ecma-404.htm.

[2] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *SIGPLAN Not.*, Vol. 37, No. 9(2002), pp. 36–47.

[3] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *SIGPLAN Not.*, Vol. 39, No. 1(2004), pp. 111–122.

[4] Green, T. J., Gupta, A., Miklau, G., Onizuka, M., and Suciu, D.: Processing XML Streams with Deterministic Automata and Stream Indexes, *ACM Trans. Database Syst.*, Vol. 29, No. 4(2004), pp. 752–788.

[5] Hosoya, H. and Pierce, B. C.: XDuce: A Statically Typed XML Processing Language, *ACM Trans. Internet Technol.*, Vol. 3, No. 2(2003), pp. 117–148.

[6] Konrad, C. and Magniez, F.: Validating XML Documents in the Streaming Model with External Memory, *ACM Trans. Database Syst.*, Vol. 38, No. 4(2013), pp. 27:1–27:36.

[7] Kuramitsu, K.: Nez Parsing Library. http://github.com/nez-peg/nez.

[8] Murata, M., Lee, D., Mani, M., and Kawaguchi, K.: Taxonomy of XML Schema Languages Using Formal Language Theory, *ACM Trans. Internet Technol.*, Vol. 5, No. 4(2005), pp. 660–704.

[9] Pierce, B. C.: *Types and Programming Languages*, MIT Press, 2002.