解析表現文法をベースにしたトランスパイラフレームワークの設計と実装

多田 拓[1,a)] 渡邉 遥輔[1,b)] 加瀬 豊[1,c)] 山口 大輔[2,d)] 倉光 君郎[3,e)]

概要：近年，センサデータや設定ファイルは JSON や XML などのデータフォーマットで記述されていることが多い．これらのデータを解析する際に F#や Scala で実装されているタイププロバイダを活用するのが便利である．タイププロバイダはデータから型を推論し，データをプログラムの型で型付けするパーサを提供する．しかし，使用できる言語限られているため，JavaScript と Web アプリケーションのように実行環境と言語が一体化しているようなケースに対して，タイププロバイダを使用することができない．我々は，マルチプラットフォームに対応したタイププロバイダの実現のため，トランスパイラによる言語非依存のタイププロバイダを提案する，本研究では，トランスパイラによる言語非依存のタイププロバイダであるトランスタイププロバイダの設計と実装を行う．

キーワード：解析表現文法，タイププロバイダ，トランスパイラ

1. はじめに

近年，センサデータや設定ファイルは JSON や XML などのデータフォーマットで記述されていることが多い．これらのデータを解析するプログラムはネストの深いパターンマッチや複雑なパーサを必要とする．タイププロバイダ [3] はデータから型を推論し，データをプログラムの型で型付けするパーサを提供する．これにより，プログラミング言語の型でデータに安全に取得することが可能になり，既存の IDE の予測機能などのサポートを受けることもできる．しかし，タイププロバイダは F#や Scala といった限られた言語でしか実装されていない．そのため，JavaScript と Web アプリケーションのように実行環境と言語が一体化しているようなケースに対して，タイププロバイダを使用することができない．我々は，マルチプラットフォームに対応したタイププロバイダの実現に向けたトランスパイラによる言語非依存のタイププロバイダ，トランスタイププロバイダを提案する．トランスタイププロバイダは形式文法の一つである解析表現文法 (Parsing Expression Grammar)[2] をベースにしている．データフォーマットを解析表現文法の拡張である TPEG によって記述し，TPEG から型推論を行うことでデータの型を得る．型の中間表現として Shape[3] を採用し，データの型をプログラミング言語の ADT や Class などで表現することを目指す．また，TEPG から生成されるパーサや推論されたプログラミン

1 横浜国立大学大学院理工学府
2 横浜国立大学大学院工学府
3 日本女子大学理学部数物科学科
a) tada-taku-jp@ynu.jp
b) watanabe-yosuke-wj@ynu.jp
c) kase-yutaka-wc@ynu.jp
d) yamaguchi-daisuke-bf@ynu.jp
e) kuramitsuk@fc.jwu.ac.jp

グ言語の型はトランスパイラによってターゲット言語に変換される．トランスパイラは抽象構文木によるルールによって柔軟にターゲット言語を切り換え可能である．本論文ではトランスタイププロバイダの概要と設計について報告する．

本論文の構成は以下の通りである．2 節では，TPEG について述べる．3 節では，型の中間表現である Shape について述べる．4 節では，トランスタイププロバイダの設計について述べる．5 節では，関連研究について述べる．6 節では，まとめと今後の展望を述べる．

2. TPEG

TPEG は解析表現文法 (Parsing Expression Grammar)[2] の拡張である．抽象構文木の構築を含めた構文定義が可能であり，構文定義から構築される抽象構文木の構造が決定できる．

2.1 文法

TPEG の文法は 5 つ組の G によって定義される．

定義 2.1 (TPEG)．G は 5 つ組 $(N_G, \Sigma, P_G, e_s, \mathcal{S})$ で定義される．ただし，N_G は非終端記号の有限集合，Σ は終端記号の有限集合，P_G は導出規則の有限集合，e_s は開始表現，$\mathcal{S}$ はラベル記号の有限集合を表す．

導出規則 $r \in P_G$ は対 (A, e) のことであり，$A \leftarrow e$ で定義される．P_G は非終点記号を解析表現に写す関数とみなされる．

解析表現 e は図 1 に定義される通りである．

TPEG は解析表現文法の解析表現を全て引き継いでいる．ここでは，非形式的な意味論について説明する．空文字列 ε は空文字列にマッチする．文字 a は同じ入力文字である終端記号 a にマッチする．非終端記号 A は，入力に対し $P_G(A)$ によるマッチを試みる．連接 $e_1\ e_2$ は，e_1 の解析表現を試し，続けて e_2 を試す．優先度付き選択 e_1/e_2 は先に e_1 の解析表現を試し，失敗した場合は e_1 の開始状態まで読み位置を戻して e_2 を試す．0 回以上の繰り返し $e*$ は正規表現の繰り返しと同様にマッチが失敗するまで貪欲に解析表現 e を繰り返し試す．否定先読み $!e$ は解析表現 e がマッチに失敗したときに全体として成功とし，e がマッチに成功したときに全体として失敗とするが，入力文字列を消費しない．

拡張された構文は capture と fold-capture であり，それぞれ ξ からツリーを構築する解析表現である．capture $\{\xi\ \#\mathbf{L}\}$ は，ξ から **L** によってラベル付けされたツリーを構築する．fold-capture $e\ (\nu = \wedge^*\{\xi\ \#\mathbf{L}\})$ は，**L** によってラベル付けされた左結合のツリーを構築する．構築される木の左側部分木は ν によってダグ付けされる．ξ は,$\nu = e$ のように e によって構築されたツリーを ν によってタグ付けしたサブツリーによって表現される．e によって構築されるツリーには終端記号のマッチによって構築されるツリーと capture や fold-capture によって構築されるツリーがある．また，$\mathsf{absorb}(e)$ は e によって構築されたツリーをサブツリーとして含まないことを示す．

例 2.2 に掛け算の式にマッチして左結合のツリーを構築する TPEG を示す．ここで，$[0-9]$ は $0/../9$ のシンタックスシュガーである．例 2.2 に入力文字列として $1*2*3$ を与えると，図 2.1 のような左結合のツリーを構築する．TEPG では左再帰の文法を禁止しているがこのように fold-capture を使用することで左結合のツリーを構築することが可能になる．

例 2.2. 掛け算の式にマッチして左結合のツリーを構築する *TPEG*

$$
\begin{aligned}
G &= (\{Val, Prod\}, \{0, 1, 2, \cdots, 9, *\}, P_G, Prod, \{\mathbf{Int}, \mathbf{Mul}\}) \\
P_G &= \{Prod \leftarrow Val\ (left = \wedge^*\{(\mathsf{absorb}(*)\ right = Val) * \#\mathbf{Mul}\}) \\
&\quad ,\ Val \leftarrow \{num = [0-9]\ \#\mathbf{Int}\}\}
\end{aligned}
$$

3. Shape

TPEG からプログラミング言語の型を提供するために，型の中間表現として Shape[3] を採用する．Shape は TPEG から推論され，トランスパイラによってターゲット言語の型に変換される．

我々は Shape をラベルに対しての型付けとして表現する．形式的には shape s で型付けされたラベル t を $t : s$ と表現する．また，この型付け文脈を Shape 文脈 E と表現する．Shape s を図 3 のように定義した．

e	::=	ε		empty
		a	$(a \in \Sigma)$	any terminal
		A	$(A \in N_G)$	any nonterminal
		$e\ e$		sequence
		e/e		ordered choice
		$e*$		zero-or-more repetition
		$!e$		not-predicate
		$\{\xi\ \#\mathbf{L}\}$	$(\mathbf{L} \in \mathcal{S})$	capture
		$e\ (\nu = \wedge^*\{\xi\ \#\mathbf{L}\})$	$(\mathbf{L} \in \mathcal{S})$	fold-capture
ξ	::=	$\nu = e$		tagged-subtree
		$\xi\ \xi$		subtree sequence
		$\mathsf{absorb}(e)$		absorption

図 1 TPEG の解析表現

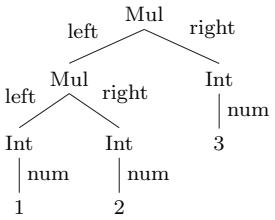

図 2 $1 * 2 * 3$ のツリー

s	::=	$\hat{s}$	non-nullable value
		$s[]$	array
		$\mathsf{nullable}\langle\hat{s}\rangle$	nullable
		null	null
$\hat{s}$	::=	$\{l_1 = s_1, \cdots, l_n = s_n\}$	recode
		$s_1 \vert \cdots \vert s_n$	common
		value	value
		t	shape variable

図 3 Shape の定義

Shape は null を持つ s (nullable shape) と null を持たない $\hat{s}$ (non-nullable shape) によって表現される．s は 4 つの shape を持ち，そのうちのひとつが $\hat{s}$ (non-nullable shape) である．$s[]$ (array) はプログラミング言語の Array や List に対応している．$\mathsf{nullable}\langle\hat{s}\rangle$ (option) はプログラミング言語の Option や Nullable に対応する．null (null) はプログラミング言語の null に対応している．$\hat{s}$ は 4 つの shape を持つ．$\{l_1 = s_1, \cdots, l_n = s_n\}$ (record) はタグ l によって束縛された s を複数持っており，プログラミング言語の Class や Recode に対応する．$s_1|\cdots|s_n$ (common) はプログラミング言語の ADT に対応する．value (value) はプログラミング言語のプリミティブな型に対応する．t (shape variable) は，ラベル t によって束縛された shape である．

以下に Shape の例を示す．

例 3.1. 例 *2.2* の *TPEG* から推論される *Shape*

$$Mul : \{left = Mul, right = Val\}$$
$$Val : Int$$
$$Int : \{num = \mathsf{value}\}$$

4. トランスタイププロバイダ

本節では，トランスタイププロバイダの設計を図 4 の模式図によって述べる．トランスタイププロバイダはターゲット言語の 3 種類のソースコードを生成する．

(1) ADT もしくは Class の定義 (ADT/Class)
(2) 抽象構文木を定義された ADT や Class に変換する処理系 (Tree to Instance)
(3) データから抽象構文木へ変換する TPEG パーサ (TPEG Parser)

3 種類のソースコードは以下の手順で取得する．まず，型を取得したいデータの構文を定義した TPEG を用意する．定義した TPEG から 3 節で述べた Shape を推論する．推論された Shpae からターゲット言語の ADT もしくは Class をトラン

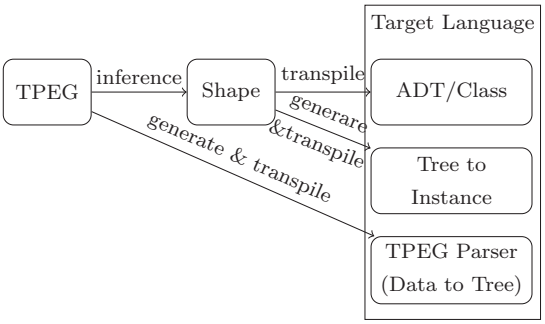

図 4 トランスタイププロバイダの模式図

スパイラで生成する．さらに，推論された Shpae から抽象構文木を定義した ADT や Class のインスタンスに変換する処理系をトランスパイラで生成する．データから抽象構文木へ変換する TPEG パーサは TPEG からパーサ生成を行い，トランスパイラによってターゲット言語に変換する．

生成されたソースコードによって以下の手順でユーザは型付けされたデータを取得できる．TPEG パーサが型を取得したいデータを入力として抽象構文木を出力する．変換された抽象構文木を定義された ADT や Class のインスタンスに変換する．

取得したインスタンスは型によって安全にアクセスすることができ，IDE の予測機能などのサポートを受ける事ができる．また，ターゲット言語は変更可能であり，所望のプラットフォームに対応した言語で生成できる．

5. 関連研究

タイププロバイダは現実世界のデータに対してプログラミング言語の型を推論して提供する．このようなデータに対して型付けを行う研究として PADS/ML[1] がある．PADS/ML の型は多くのパターンをサポートしており，表現力が高い．また，独自のフォーマットは簡潔に記述可能であり，OCaml のコードへコンパイルされる．本研究のアプローチは TPEG によって文法開発者が慣れ親しんでいる構文でフォーマットを記述できる．これに対し，F#Data のタイププロバイダ [3] は提供される JSON や XML などのデータパーサにデータを与えて型付けを行う．これはユーザがフォーマットの定義をしなくてよい反面，用意されていないデータフォーマットに対して使用できない．この点では，本論文で提案した TPEG をベースにした実装は柔軟なフォーマットの対応が可能であると言える．

既存のタイププロバイダは限られた言語のみでの実装になっており，プラットフォームと言語が一体化しているようなケースに対しての使用が難しい．本論文ではマルチプラットフォームで動作可能なタイププロバイダを提案した．

6. まとめと今後の展望

本論文では言語非依存はタイププロバイダの概要と設計について報告した．解析表現文法の拡張である TPEG をベースとした設計とし，TPEG から Shape への推論とパーサ生成を行う．生成されたターゲット言語のソースコードは，データをパースし，推論された型の ADT もしくは Class のインスタンスに変換する．これにより，マルチプラットフォームでの動作を可能とする．

今後の展望は，本論文で提案したタイププロバイダの実装である．パラダイムの異なる言語をより多くサポートし，より多くのプラットフォームでの動作を目指す．

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*, POPL '08, New York, NY, USA, ACM, pp. 421–434 (online), DOI: 10.1145/1328438.1328488 (2008).

[2] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, New York, NY, USA, ACM, pp. 111–122 (online), DOI: 10.1145/964001.964011 (2004).

[3] Petricek, T., Guerra, G. and Syme, D.: Types from data: Making structured data first-class citizens in F#, *Proceedings of Conference on Programming Language Design and Implementation*, PLDI 2016.