型検査を用いたコンパイル時 LR 構文解析手法の提案

松永 智將　市川 和央　山崎 徹郎　中丸 智貴　千葉 滋

本論文は，コンパイル時に LR オートマトンをエミュレートする手法を提案する．本手法を用いると，LR 文法を持った fluent API の文法をコンパイル時にチェックできる．本手法では、型検査器を用いて文法検査をする．つまり，LR 構文解析の現在の状態とスタックの中身を型で表し，入力トークンを関数で表す．入力トークンを読んだ時の LR 構文解析の状態遷移を，その関数の呼び出しに対応づける．入力トークン列が文法的に正しければ，この関数呼び出しは型付けできる．このような型付けが可能な言語機構として C++ template や型クラスがあることを述べる．

1　はじめに

DSL (Domain Specific Language) とは，特定のドメインの問題解決を目的として設計された言語である．特に，内部 DSL (Embedded Domain Specific Language) は、汎用プログラミング言語の内部に記述可能な DSL を指す．内部 DSL は，ホスト言語 に依存するため文法の制限を受けるが，ホスト言語や他のライブラリとの連携がしやすいという利点を持つ．

静的型付き言語では，型を利用することで内部 DSL のプログラムの簡単な誤りを検出することができる．本論文では，型情報を利用してコンパイル時に LR 文法相当の文法検査を行う方法を提案し，そのために必要な言語機構を明らかにする．また，それらの言語機構を持つプログラミング言語での実装方法を紹介する．

2　内部 DSL と静的エラー検出

内部 DSL を設計するための手法の 1 つとして fluent API がある [1]．fluent API はトークンをメソッドチェインスタイルで記述することで内部 DSL を表現する．また Scala などの言語はメソッド呼び出しの際のドットや括弧の省略記法を持っており，これを利用することでより DSL らしい文法を実現することができる．

```
var query: Query =
  select("*")
  .from("posts")
  .where("id").eq("1")
  .build()
```

図 1　正しい SQL クエリの例

内部 DSL も一種のプログラミング言語であるため，コンパイル時に不正なプログラムを検出できることが望ましい．fluent API による内部 DSL の場合、メソッドチェイン中のメソッド呼び出しの並びが不正であるものをエラーとして検出したい．例として，図 1 のような SQL を fluent API で表現することを考える．2 章のコード例は全て Nim というプログラミング言語で記述している．ここで，図 2 のような SQL の文法として正しくないメソッドの並びはコンパイル時にエラーにしたい．

型システムを利用することで，簡単なエラーであれば検出することができる．図 2 の文法エラーを型システムを利用して検出するには，図 3 のように型と関数を定義する．図 3 では，はじめにオートマトンの状態を表すクラスが宣言されている．例えば，Select はプ

Tomomasa Matsunaga, Kazuhiro Ichikawa, Tetsuro Yamazaki, Tomoki Nakamaru, Shigeru Chiba, 東京大学, The University of Tokyo University.

```
var query: Query =
  select("*")
  .where("id").eq("1")
  .from("posts")
  .build()
```

図 2　不正な SQL クエリの例

```
type
  Select = ref object of RootObj
    val: string
  From = ref object of RootObj
    val: string

proc select(val: string): Select = Select(val:
      val)
proc `from`(n: Select, val: string): From =
     From(val: val)
```

図 3　文法エラーを検出可能な DSL 定義の例

ロパティとして string 型の val を持つクラスである．以降の行では実際に fluent API で使用される関数が宣言されている．select 関数は string 型の引数 val をとり，Select オブジェクトを返す関数である．同様に from 関数は Select 型の n と string 型の val を引数にとる．この例では Nim の Uniform Function Call Syntax(UFCS) という言語機構を用いて fluent API のメソッドチェーンを表現している．UFCS とは、関数呼び出しをその関数の第一引数をレシーバとしたメソッド呼び出しのように記述するための言語機構である．これにより，from 関数は Select オブジェクトをレシーバとしたメソッドだと捉えられる．select 関数の返り値は Select オブジェクトなので，以降に続く関数を from 関数のみに制限することができる．

しかし，この手法を単純に適用すると，複雑な例の場合メソッド定義の個数が多くなりすぎてしまう．例として，図 4 のようにこの fluent API をサブクエリに対応させることを考える．図 3 のように各関数ごとに 1 つのクラスのみを定義しておく場合，図 5 のように括弧が対応していない不正な並びを検出することができない．これは，サブクエリの eq 関数の返り値の型が，メインクエリの eq 関数の返り値の型と

```
var query: Query =
  select("*")
  .from("posts")
  .where("user_id").in
    .lp
      .select("id")
      .from("users")
      .where("name").eq("foo")
    .rp
  .build()
```

図 4　サブクエリを含んだクエリの例

```
var query: Query =
  select("*")
  .from("posts")
  .where("user_id").in
    .lp
      .select("id")
      .from("users")
      .where("name").eq("foo")
  .build()
```

図 5　不正なサブクエリを含むクエリの例

同じであることが原因である．この問題を解決するために，ネスト深さに対応したクラスを定義しておくことが考えられるが，これを深さごとに全て定義しておくことは不可能である．

3　型検査を用いたコンパイル時 LR 構文解析

本論文では，型検査を用いたコンパイル時 LR 構文解析の手法を提案する．コンパイル時にメソッド呼び出しの並びが不正であるものを検出するという問題は，各メソッド呼び出しをトークンだと考えると構文解析の問題に帰着する．そのため，既存の構文解析手法を適用することができれば，より形式的な手法で内部 DSL のエラーを検出することが可能になる．本論文では，LR 構文解析を既存言語に存在する言語機構により再現する手法を示す．また，それらの言語機構を持つプログラミング言語での実装方法を紹介する．簡単のために，本章では図 4 を単純化した select， from のみの SQL を扱うものとする．

3.1 LR 構文解析の再現手法

LR 構文解析は決定性有限オートマトンをベースとした構文解析手法で，shift と reduce という 2 つの操作により構文解析を行う．shift はトークンを 1 つ読み，オートマトンの状態遷移を行う．reduce はオートマトンの遷移を巻き戻し， 別の状態に遷移する．この reduce 操作がこの構文解析が LR 構文解析と言われる所以で，入力を左から読み (**L**eft-to-right)，構文規則の右辺から左辺の非終端記号を導出する (**R**ightmost derivation) ことを表現している．以上から， 型検査で LR 構文解析を再現するためには shift と reduce の操作を再現すれば良い．

入力トークン列はメソッド呼び出しの並びなので，メソッド呼び出しの型で状態遷移を表現することができる．例えば， 以下の関数定義を考える．

```
proc `from`(n: Select, val: string): From =
    From(val: val)
```

これは from 関数を定義しているが， Nim の UFCS により Select クラスのメソッドのように振舞うことができる．そのため， 以下のように使うことが可能である．

```
var s: Select = ...
var x: From = s.from("posts")
```

それぞれの型がオートマトンの状態を表現していると考えると， メソッド from は状態 Select から状態 From への状態遷移を表現している．

LR 構文解析でトークンを 1 つ読んだ際に実行されるのは， 0 回以上の reduce 操作と 1 回の shift 操作であるため，各メソッドでは 0 回以上の reduce 操作と 1 回の shift 操作を合成した操作を再現する必要がある．しかし，当然ではあるが reduce の回数によって状態遷移の仕方が変わるため，それらをメソッドのオーバーロードによって表現しなければならない．reduce 操作の回数の上限がわかっている場合は，全てのパターンのメソッドをオーバーロードして定義すればよい．しかし，文法規則に右再帰が含まれる場合は reduce 操作の回数の上限がなく，無限パターンのメソッドを定義しなければならない．

したがって，本論文では無限個のメソッドのオーバーロードを行うことができる仕組みを利用してコンパイル時に LR 構文解析を行う手法を提案する．無限個のメソッドのオーバーロードを行うことができる仕組みとして，本論文ではテンプレート関数と型推論の組み合わせによる解法と，型クラスによる解法の 2 つを紹介する．

3.2 テンプレート関数と型推論による手法

テンプレート関数を用いて，shift 操作と reduce 操作を合成した全ての遷移を表す関数をオーバーロードして定義することができる．本節では，テンプレート関数と型推論を使えるプログラミング言語として Nim 言語を利用する．遷移を表すメソッドはレシーバにあらゆる型を取ることのできるテンプレート関数として定義する．このテンプレート関数は，shift 操作が可能な場合は shift した結果を返し，それ以外の場合は，reduce した結果を再帰的に遷移させる．すなわち，0 回以上の reduce 操作と 1 回の shift 操作を合成した遷移を表す．不正なメソッド呼び出し列が入力された場合は，展開後のコードが不正な呼び出しを行うので型エラーとして検出できる．遷移を表す関数の例は以下のようになる．

```
proc select[N](n: N, id: string): auto =
  n.reduce().select(id)

proc select[N](prev: Node3[N], id: string):
    Node1[Node3[N]] =
  Node1[Node3[N]]()
```

1 つ目の定義が，reduce した結果を再帰的に遷移させるテンプレート関数である．select 関数のボディで呼ばれている reduce 関数はオーバーロードされており，型 N によって異なる型の値を返す．そのため，select 関数の返り値の型は記述が不可能な型となっている．しかし、返り値の型を auto とすることでコンパイラがテンプレートの展開を行う際に N の型それぞれについて解決させることができる．2 つ目の定義は，shift 操作が可能な場合の特殊化である．

以下に reduce 操作を表す関数の例を示す．1 つ目は，ネストした select 文の最も内側の select 文の reduce 操作を表す．2 つ目は，select 文全体を受理状態へ遷移させる reduce 操作を表す．これらは，LR

構文解析の構文解析表から自動的に導出できる．

```
proc reduce[N](n: Node2[Node1[Node3[N]]]):
    Node4[Node3[N]] =
  Node4[Node3[N]]()

proc reduce(n: Node4[Node3[Node1[Node0]]]):
    Node5[Node0] =
  Node5[Node0]()
```

3.3 型クラスによる手法

shift 操作と reduce 操作及び、それらを合成した遷移を型クラスとして表すことで，LR 構文解析を再現する．本節では、型クラスを使えるプログラミング言語として Scala を利用する．shift 操作，reduce 操作及び遷移は図 3.3 のようにして，宣言することができる．はじめの 3 つの case クラスの宣言が型クラスの宣言である．それに続く，implicit 関数 shift_transition は 1 つの shift 操作が遷移を表現することを表す．また，implicit 関数 reduce_transition は複数の reduce 操作と遷移を組み合わせても遷移となることを表している．

Trans[T, N1, N2] は遷移を表すので，Trans[T, N1, N2] のオブジェクトが存在するとき，対応する遷移メソッドを呼ぶことができる．これを表現するのが以下の implicit class の定義である．

```
implicit class T_Select [N1, N2] (node: N1)
  (implicit t: Trans[Select.type, N1, N2]){
  def select: N2 = t.transit(node, Select)
}
```

この定義は N1 から Select を読むことによる N2 への遷移を表す．Trans[Select.type, N1, N2] の implicit parameter を取得できるということが遷移可能であることに対応するので，この時 N1 のメソッドとして N2 へ遷移する select メソッドが定義される．

shift 操作と reduce 操作はそれぞれ implicit 関数として定義する．implicit parameter として要求された値の型が，implicit 関数の返り値の型と一致する時に自動的に呼び出される．これにより，図 3.3 と合わせてコンパイル時に暗黙的に Trans オブジェクトが生成される．以下が shift 操作と reduce 操作の定義の例である．

```
implicit def shift_node1_id [N]:
  Shift[Id, Node1[N], Node2[Node1[N]]]
  = Shift((s, t) => Node2(s, t.value))
implicit def reduce_node4 [N]:
  Reduce[Accept.type, Node4[N], Node5[N]]
  = Reduce(s => Node5(s.prev, s.value))
```

我々はこの手法を利用した LR 構文解析系である ScaLALR を開発した．ソースコードはオープンソースとして GitHub に公開している †1．

4 関連研究

Gil らは文献 [2] において，Java の型検査を用いたコンパイル時構文解析の手法を提案している．Gil らの手法は本論文での手法と同じく LR 文法に対して適用することができるが，実現方法は大きく異なる．Java の型システム上では，LR 構文解析器を素朴には再現できないため，Gil らは jump-stack real-time deterministic pushdown automaton (JRDPDA) を利用している．しかしながら Gil の手法では，型検査にかかる時間が最悪の場合チェイン長に対して指数的に増大するという問題があった．一方，本論文の手法ではこのような型検査の指数的増大は発生しない．EriLex [5] や Silverchain [4] でも，型検査にかかる時間の指数的増大は起きない．しかしながら対応できる文法クラスは，本手法の対応する LR 文法に比べて非常に小さいという問題がある．

理論的には，型システムがチューリング完全であれば任意の文法に対してコンパイル時の構文解析ができる．例えば Java の型システムはチューリング完全である [3]．つまりチューリングマシンを Java の型定義によって構築できる．このチューリングマシンの上に，CYK パーサなど，より広い範囲の文法を取り扱う構文解析器を再現すればコンパイル時構文解析は可能となる．しかしながらこの方法は非常に大きな計算リソースを要し，多くの内部 DSL にとっては過度に強力な手法である．一方，本論文の手法では計算リソースの極端な利用はない．

†1 https://github.com/phenan/scalalr

```
case class Shift [T, N1, N2] (shift: (N1, T) => N2)
case class Reduce [T, N1, N2] (reduce: N1 => N2)
case class Trans [T, N1, N2] (transit: (N1, T) => N2)

implicit def shift_transition [T, N1, N2]
  (implicit shift: Shift[T, N1, N2]): Trans[T, N1, N2]
   = Trans { shift.shift }
implicit def reduce_transition [T, N1, N2, N3]
  (implicit reduce: Reduce[T, N1, N2], trans: Trans[T, N2, N3]): Trans[T, N1, N3]
  = Trans { (s, t) => trans.transit(reduce.reduce(s), t) }
```

図 6　shift と reduce を表現する型クラス

5　まとめ

本論文では，型情報を利用してコンパイル時に LR 文法を持った内部 DSL の文法チェックを行う方法を提案した．それを実現する手法として，テンプレート関数と型推論の組み合わせによる手法と型クラスによる手法を紹介した．

参 考 文 献

[1] Martin Fowler. Fluentinterface, December 2005.

[2] Yossi Gil and Tomer Levy. Formal Language Recognition with the Java Type Checker. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Vol. 56, pp. 10:1–10:27, 2016.

[3] Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pp. 73–85, New York, NY, USA, 2017. ACM.

[4] Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: A Fluent API Generator. In *GPCE 2016: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2017.

[5] Hao Xu. *EriLex: An Embedded Domain Specific Language Generator*, pp. 192–212. Springer, 2010.