Fluent APIにおけるプログラム断片のより柔軟な結合

中道 晃平　中丸 智貴　森畑 明昌

Fluent API はメソッド呼び出しの連鎖によって埋め込み DSL を実装するデザインパターンの一つで，ホスト言語の型検査を利用してゲスト言語の構文検査をする研究がなされている．本論文では DSL のトークンをメソッドの連鎖で繋ぐのではなく，演算子を利用して繋ぐ fluent API のデザインを提案する．演算子で繋ぐデザインによって，DSL の断片のより簡潔な構築と結合が可能になる．しかし，既存のプログラム断片の型付けを行う fluent API 設計手法では，プログラマにとって自然なプログラムが書けないことがある．そこで本論文では既存手法の 1 つである，DSL の文法を変形することによりプログラム断片の型付けを可能にする手法（Yamazaki ら，SLE2022）を拡張することより，より柔軟なプログラム断片の組み合わせを可能にする Fluent API の設計が可能であることを示す．

1　はじめに

Fluent API は，ゲスト言語のプログラムをメソッドの連鎖によってホスト言語中に記述するという，埋め込み DSL のデザインパターンである．図 1 は Fluent API によって C++に下記の SQL プログラムを埋め込んだ例である．

```
SELECT *
FROM BOOK
WHERE PRICE <= 2000
```

この例では，SQL のキーワード名を名前とするメソッドをドット演算子を用いて連鎖的に呼び出している．ドット演算子やメソッド呼び出しの括弧等を無視すれば，これは SQL プログラムをそのまま書いているかのような記述となっている．

Fluent API には，ホスト言語においてメソッドの連鎖が可能であることだけが求められる．そのため，広い範囲のホスト言語・ゲスト言語に対して利用可能である．実際に fluent API を利用しているライブラリとしては，JOOQ[†1] や SuperTest[†2] などが挙げられる．

```
Query query = Query
 .select().column("*")
 .from().table("BOOK")
 .where().column("PRICE").le().num(2000);
query.run();
```

図 1: Fluent API を利用した SQL 埋め込みの例

Fluent API の欠点としては，なんの工夫もしなかった場合，ゲスト言語のプログラムとして全く意味をなさないメソッド連鎖が容易に記述できてしまうという点が挙げられる．例えば，図 2 のコード例では，`select` が連続して出現しているため，SQL の構文には合致しない．このような構文的な誤りを静的に発見できれば，プログラムの実行前にプログラマがバグに気づくことができ，開発効率が向上する．

そこで近年，fluent API で記述された埋め込み DSL のプログラムやその断片が，ゲスト言語の構文規則に合致しているかどうかを，型検査を利用してチェックする研究がなされている [4] [7] [9] [10] [11]．例えば，Gil らの手法 [4] では決定性文脈自由言語を対象にし

Flexible concatenation of program fragments in fluent API

Kohei Nakamichi, Tomoki Nakamaru, Akimasa Morihata, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, The University of Tokyo.

†1 https://www.jooq.org/

†2 https://github.com/ladjs/supertest

```
Query query = Query
 .select().column("*")
 .select().column("*")
 .from().table("BOOK")
query.run();
```

図 2: SQL の構文的に正しくない記述の例

```
From origin = from() + table("BOOK")
Where cond =
  where() + column("PRICE") + le() + num
      (2000);
Query query =
  select() + column("*") + origin + cond;
```

図 3: 提案手法による SQL 埋め込みの例

て，決定性プッシュダウンオートマトンを型システムを用いてシミュレートすることで構文検査を実現している．型システムによる構文検査では，大量の型やメソッドの定義が必要になる．そのため，これらの研究の多くは，文法を入力として構文検査のためのプログラムを出力する fluent API 生成器の作成を目指している．

本研究では，メソッド連鎖を用いて DSL のトークンを 1 つずつで繋ぐのではなく，**演算子を利用してプログラム断片同士を繋ぐ** fluent API のデザインを提案する．例として、図 3 に，提案手法を用いて図 1 のプログラムを記述したものを示す．プログラム断片は+演算子を用いて結合される．

この方式では，メソッド連鎖を利用する既存方式に比べ，プログラムを意味上にまとまった部分ごとに構築し，それらを結合して大きなプログラムを構成してゆくようなプログラミングが，より自然に表現できる．例えば図 3 のプログラムであれば，データを取り出すテーブルや取り出されるレコードを特定する条件が，クエリ全体とは独立したプログラム断片として定義できている．そのため，fluent API を用いたプログラム記述におけるモジュール性を改善すると期待できる．

Fluent API において，プログラム断片を組み合わせることをある程度許す方式についても研究がなされている [7] [11]．しかし，これら既存手法においては，どのようなプログラム断片の組み合わせが結合可能であるかは，DSL の仕様として与えられる文脈自由文法に強く制限される．そのため，プログラマにとってはプログラム断片の結合をするプログラムが書けないことがある．

```
From origin = from() + table("BOOK")
ColumnList cols1 =
  column("PRICE") + column("PRODUCT_ID");
ColumnList cols2 =
  column("QUANTITY") + column("NAME");
ColumnList col3 = cols1 + cols2;
Query query = select() + cols3 + origin;
```

図 4: 既存手法では構文検査に失敗する例

一例を図 4 に示す．このプログラムでは，抽出したい列名を `col1` と `col2` という 2 つの部分に分けて構成し、それらを結合して `col3` を構成している。一見自然だが，既存手法に基づいて構文検査をしようとすると，構文エラーと判定されてしまう．

本研究では，上記の問題が既存研究 [10] の自然な拡張で解決できることを示す．この手法は，Fluent API 生成器へ入力する文法を事前に変形しておくものである．そのため，型システムによる構文検査に際しては，既存手法をそのまま流用することが可能である．

本研究の主な貢献は次の通りである．

- 演算子オーバーロードを利用した，演算子で埋め込み DSL のトークンを繋ぐデザインの fluent API を提案する．
- プログラマにとって直感的な記述であるにもかかわらず，既存手法では扱うことのできないプログラム断片の結合が存在することを示す．
- 既存研究の上記の問題点が，文法を適切に変形することによって解消できることを示す．

2 背景

2.1 Fluent API における構文検査

Fluent API における構文検査では，メソッドをトークンとみなす．正しいメソッドの連鎖であるとは，メソッドの連鎖をトークン列とみなしたときにそ

のトークン列がゲスト言語の文法仕様を満たしているということである．

構文検査手法の例として，Yamazaki らの手法[11]の概要について説明する．C++の型システムを利用して LR オートマトン[3] のシミュレートをする．

LR オートマトンの状態は型によってエンコードされる．LR オートマトンの状態におけるスタックは，`S1<S2<Bottom>>`のようにテンプレート引数を入れ子にすることで表現できる．

オートマトンの状態遷移はメソッド呼び出しによって行われる．メソッド連鎖の開始は，オートマトンの開始状態に対応する型をもつオブジェクトである．オブジェクトからのメソッド呼び出しにより，オートマトンの遷移が引き起こされる．遷移後の状態は，メソッドの返り値の型として返される．メソッド呼び出しの連鎖によりオートマトンが状態遷移していくことで，トークン列が文法仕様を満たしているかが検査される．トークン列が文法仕様を満たしていないときは，メソッド呼び出しは型検査に失敗する．

2.2 Sub-chainining style

Fluent API に対する静的な構文検査の研究の多くは，切れ目や入れ子のない，ひと繋がりのメソッド連鎖を対象にしている．このような fluent API は flat-chainining style と呼ばれる，

ゲスト言語のプログラム断片を作成し，断片同士を組み合わせることのできるデザインの fluent API についても研究がなされている[7][11]．このような方式は，flat-chainining style に対して，sub-chainining style と呼ばれる，

図 5 と図 6 は同じ意味のコードをそれぞれ flat-chainining style と sub-chainining style で記述したものである．どちらも，SQL クエリの where 節が条件変数 `b` の値によって変化する．Flat-chainining style では，不完全なメソッド連鎖を一旦変数 `tmp` に束縛し，`if` 文の中でこれに where 節に対応するメソッド連鎖を付け加える．一方で sub-chainining style では，where 節に対応するメソッド連鎖を一度変数 `cond` に束縛して，それを用いてクエリ文全体を組み上げる．

このように，flat-chainining style ではのプログラム全体を先頭から順に作成する必要があるのに対し，sub-chainining style ではプログラムを一部分ずつ作って組み合わせる記述が可能になる．そのため，sub-chainining style のほうがモジュール性に優れるといえる．

```
IncompleteChain tmp = Query
  .select().column("*")
  .from().table("BOOK");
Query query;
if(b) {
  query =
  tmp.where().column("PRICE").le().num(2000);
} else {
  query =
  tmp.where().column("PRICE").ge().num(2000);
}
```

図 5: Flat-chaining style のコード例

```
Where cond;
if(b) {
  cond = Where
    .where().column("PRICE").le().num(2000);
} else {
  cond = Where
    .where().column("PRICE").ge().num(2000);
}
Query query = Query
  .select().column("*")
  .from().table("BOOK")
  .where(cond);
```

図 6: Sub-chaining style のコード例

2.3 Yamazaki らによる sub-chaining style の実現手法

Sub-chainining style を実現する手法として，本研究と関わりの深い Yamazaki らの手法[10] を紹介する．

Yamazaki らの手法では，まず文法仕様として与えられた文法の変形を行う．文法の変形においては，各非終端記号に対してその非終端記号から導出される

ようなメソッド列を受け取るためのメソッドを追加する．図 7 を変形したものを図 8 にを示す．赤字が追加されたメソッドを示している．`Select`(*Select*) は *Select* から導出されるメソッド列の断片を受け取る，`Select` という名前のメソッドを表している．

そして，メソッド列の断片が各非終端記号から導出されるすることができるかをチェックするために，各非終端記号を開始記号として既存の flat-chaining style の手法で構文検査を行えるようにする．メソッド列がある非終端記号から導出可能なときは，開始記号となる非終端記号を指定して構文検査をおこなうことができる．構文検査が行われたトークン列には，開始記号とした非終端記号に対応する型がつく．これによって，文法変形の際に追加したメソッドが受け取るメソッド列が，対応する非終端記号から導出されるかをチェックすることができる．

3 提案：演算子オーバーロードを利用した fluent API

本研究では埋め込み DSL のトークンをメソッド連鎖で繋ぐのではなく，演算子で繋ぐスタイルの fluent API を提案する．

図 9 は演算子で繋ぐスタイルの fluent API のプログラム例である．既存手法ではメソッド連鎖で DSL のトークン結合していたところを，二項演算子`+`によって結合している．さらに，単項演算子`~`も利用している．ゲスト言語において括弧で囲む必要のある部分を，ホスト言語の括弧で囲み，単項演算子`~`を適用している．単項演算子`~`が必要なのは，括弧を付ける部分がホスト言語の式において先頭の場合にホスト言語側で括弧がついているかどうか識別が困難になるからである．

演算子で繋ぐスタイルの fluent API ではトークン列の断片を変数で束縛したり，関数で返したりする記述も可能である．図 10 のプログラムは図 6 のプログラムを，演算子で繋ぐデザインで書き換えたものである．トークン列の断片が束縛された変数である `cond` を結合する，最後の行の記述が特に異なっている．図 6 のプログラムではトークン列の断片を受け取る専用のメソッドを用いる必要があった．図 10 のプログラムは `cond` を直接演算子で繋いでいる．

演算子で繋ぐスタイルの fluent API ではメソッド連鎖で繋ぐスタイルの fluent API と比較して利点が 2 つある．

1 つめは演算子を利用することで，埋め込み DSL に括弧を可読性の高い形で導入することが可能になることである．メソッド連鎖を利用する fluent API に括弧を導入した場合が図 11 である．図 9 と同じクエリを記述しているが，括弧として `begin`, `end` を利用している．多くのプログラミング言語ではメソッド名として `()` などの記号を利用できない．そのため，メソッド連鎖を利用する fluent API では括弧を表現するためにアルファベットの列を利用する必要がある．しかし，アルファベットの列で表現された括弧はひと目で判別することが困難である．括弧はプログラミング言語の構文において曖昧性の除去などの重要な役割を果たす．括弧の視認性が高いことは大きな利点となる．

2 つめは，演算子で繋ぐスタイルではトークン列の部分列を結合する記述がシンプルになるとことである．図 6 のプログラムより，図 10 のプログラムのほうがトークン列の結合はシンプルであった．演算子で繋ぐスタイルでは，ゲスト言語のトークンがホスト言語において第一級市民であるためにシンプルに結合することが可能となっている．トークン列を繋ぎ合わせて全体を構成していくという観点で，演算子を用いる方がメソッド連鎖を用いるより直感的な記述が可能となる．

4 問題点とその解決

Sub-chaining style を用いることで埋め込み DSL のトークン列を結合させて記述するプログラムを自然に書くことができる．しかし，既存の sub-chaining style を実現する手法では，文法の曖昧性が問題となり，プログラマにとって自然な記述ができないことがある．本節ではそのようなプログラム例を示すと同時に，その解決策を示す．

Query ::= *Select From Where*
Select ::= `select() column(string)`
From ::= `from() table(string)`
Where ::= `where() column(string)` *Op* `num(int)`
Op ::= `le()` | `ge()` | `eq()`

図 7: SQL の文法例

Query ::= *Select From Where*
Select ::= `select() column(string)`
| `Select(`*Select*`)`
From ::= `from() table(string)`
| `From(`*From*`)`
Where ::= `where() column(string)` *Op* `num(int)`
| `where(`*Where*`)`
Op ::= `le()` | `ge()` | `eq()`
| `Op(`*Op*`)`

図 8: 文法変形の例．赤字が変形により追加された終端記号である．

```
Where cond;
if(b) {
  cond = where() + column("PRICE") + le() +
       num(2000);
} else {
  cond = where() + column("PRICE") + ge() +
       num(2000);
}
Query query = select() + column("*") + from()
     + table("BOOK") + cond;
```

図 10: 演算子を用いてトークン列の断片を組み合わせるプログラム例

```
Query query =
    select() + column("*")
  + from() + table("BOOK")
  + where() + column("PRICE") + eq() + ~(
        select() + max("PRICE")
      + from() + table("BOOK")
);
```

図 9: 演算子を用いた fluent API の記述例．サブクエリの記述をおこなっている．

```
Query query =
    select().column("*")
   .from().table("BOOK")
   .where().column("PRICE").eq().begin()
        .select().max("PRICE")
        .from().table("BOOK")
   .end();
```

図 11: メソッド連鎖を用いたサブクエリの記述例

4.1 非終端記号の列から導出可能なトークン列

4.1.1 動機となる例

既存の sub-chaining style を実現する手法 [7] [10] では，1 つの非終端記号から導出可能なトークン列にしか型検査を行うことができない．しかし，複数の非終端記号から導出されるようなトークン列を扱いたい場合がある．

図 12 は SQL の select 節と from 節を同じ変数に束縛している例である．変数 `target` に束縛されるトークン列が，非終端記号の列 *Select From* から導出されることが保証されている．

これを既存手法で実現するには，図 7 の文法に非終端記号 *SelectFrom* を追加し，図 13 のような文法にすることが考えられる．この文法では，非終端記号 *SelectFrom* によって非終端記号の列 *Select From* から導出されるトークン列を表現できる．

複数の非終端記号から導出されるようなトークン列を扱う例としては図 14 のようなプログラムも考えられる．このプログラムでは変数 `rows` に from 節と

```
SelectFrom target;
if(b) {
    target = select() + column("QUANTITY")
           + from() + table("ORDER");
    } else {
    target = select() + column("STOCK")
           + from() + table("BOOK");
}
Query q = target + where() +
    column("PRODUCT_ID") + eq() + num(42);
```

図 12: Select 節と From 節を同じ変数に束縛するプログラム

$$
\begin{aligned}
\mathit{Query} &::= \mathit{SelectFrom}\ \mathit{Where} \\
\mathit{SelectFrom} &::= \mathit{Select}\ \mathit{From} \\
\mathit{Select} &::= \texttt{select() column(string)} \\
\mathit{From} &::= \texttt{from() table(string)} \\
\mathit{Where} &::= \texttt{where() column(string)} \\
&\quad\ \mathit{Op}\ \texttt{num(int)} \\
\mathit{Op} &::= \texttt{le()} \mid \texttt{ge()} \mid \texttt{eq()}
\end{aligned}
$$

図 13: 非終端記号 *SelectFrom* を追加した文法の例

where 節を束縛している．

このプログラムを既存手法で実現するために，*SelectFrom* の場合と同じように，非終端記号 *FromWhere* を文法に追加することが考えられる．

しかし，このアプローチでは非終端記号 *SelectFrom* と *FromWhere* を共存させた場合に文法が曖昧になってしまうという問題が発生する．なぜなら，非終端記号 *Query* から *Select From Where* を導出する過程として，

Query ⟶*SelectFrom Where* ⟶*Select From Where*
Query ⟶*Select FromWhere* ⟶*Select From Where*

の 2 つが考えられるからだ．

Fluent API の構文検査を実現する既存手法では曖昧な文法を扱うことはできない．そのため上記の例のような複数の非終端記号から導出されるようなトークン列を扱いたい場合では既存手法が不十分である．

```
FromWhere rows;
if(b) {
    rows = from() + table("ORDER")
         + where() + column("QUANTITY") + le()
              + num(10);
    } else {
    rows = from() + table("ORDER")
         + where() + column("STOCK") + le() +
             num(10);
}
Query q = select() + column("PRODUCT_ID") +
     rows;
```

図 14: From 節と Where 節を同じ変数に束縛するプログラム

4.1.2 解決法

前節では 2 つ以上の非終端記号から導出可能なトークン列を，複数種類扱うことを考えた．しかし，これをナイーブに行おうとすると文法が曖昧になってしまうことがあった．

この問題は 2.3 節で紹介した Yamazaki らの手法を拡張することで解決できる．Yamazaki らの手法における文法の変形をするときに，図 8 の代わりに図 15 のように変形する．ハット（^）がついた記号は，その型がついたトークン列の断片を表す終端記号である．例えば，$\widehat{\texttt{Select}}$ は *Select* から導出可能なトークン列や，そのようなトークン列が束縛された変数を表している．†3

図 15 では図 13 と異なり，*Query* から *SelectFrom* は導出されない．その代わりに *SelectFrom* に対応する終端記号である $\widehat{\texttt{SelectFrom}}$ を導出できる．*SelectFrom* から導出されるトークン列の断片を結合するときに，文法上では非終端記号 *SelectFrom* を介さずに結合可能となる．この変形によって，*Query* から導出される記号列が自由度上がりすぎないようになっている．

一般に $A \longrightarrow X_1 \ldots X_n$ という形の文法規則があ

†3 $\widehat{\texttt{Select}}$ は，図 8 における `Select(`*Select*`)` に対応する．トークンの結合に演算子オーバーロードを用いるのに合わせて記法を変更している．

$$
\begin{aligned}
\mathit{Query} &::= \mathit{Select}\ \mathit{From}\ \mathit{Where} \\
&\mid \widehat{\texttt{SelectFrom}}\ \mathit{Where} \\
&\mid \mathit{Select}\ \widehat{\texttt{FromWhere}} \\
\mathit{SelectFrom} &::= \mathit{Select}\ \mathit{From} \\
\mathit{FromWhere} &::= \mathit{From}\ \mathit{Where} \\
\mathit{Select} &::= \texttt{select()}\ \texttt{column(string)} \\
&\mid \widehat{\texttt{Select}} \\
\mathit{From} &::= \texttt{from()}\ \texttt{table(string)} \\
&\mid \widehat{\texttt{From}} \\
\mathit{Where} &::= \texttt{where()}\ \texttt{column(string)} \\
&\quad\ \mathit{Op}\ \texttt{num(int)} \\
&\mid \widehat{\texttt{Where}} \\
\mathit{Op} &::= \texttt{le()} \mid \texttt{ge()} \mid \texttt{eq()} \\
&\mid \widehat{\texttt{Op}}
\end{aligned}
$$

図 15: *SelectFrom* と *FromWhere* を共存させる文法. 図 8 との相違点を赤い文字で示している

り，$X_i \dots X_j$ $(i \leq j)$ から導出されるトークン列の構文検査を行いたいとする．このとき

- 非終端記号
  - Y
- 終端記号
  - $\widehat{\texttt{Y}}$
- 生成規則
  - $Y \longrightarrow X_i \dots X_j$
  - $A \longrightarrow X_1 \dots X_{i-1}\ \widehat{\texttt{Y}}\ X_{j+1} \dots X_n$

を追加することで文法が曖昧になることを回避することができる．変形前の文法から生成される言語は，変形後の文法で生成される言語の部分集合となっている．また，変形前の文法が曖昧でないなら変形後の文法も曖昧でない．

4.2 繰り返しを含む文法

4.2.1 動機となる例

図 7 の文法を拡張した文法である図 16 の文法を考える．図 7 の文法では select 節に行を 1 つしか記述することができなかったが，図 16 では非終端記号 *ColumnList* によっていくつか並べたものを記述することができる．

図 17 は図 16 の文法をもとにした複数のカラムを並べるプログラムである．1 行目と 2 行目は，それぞれ変数 `columns1` と `columns2` にカラムを並べたものを束縛してる．3 行目は `columns1` と `columns2` を連結して `columns3` に束縛している．`columns3` は `ColumnList` の型がつけられている．これには次のような意図がある．

ColumnList は `column(string)` が 0 個以上連結したものを表している．したがって，*ColumnList* から導出される `columns1` と `columns2` を連結しても，`column(string)` が 0 個以上連結したものとなるはずである．そのため，`columns3` の型は `ColumnList` となっている．

しかし，図 16 の文法では *ColumnList* から *ColumnList ColumnList* を導出することができないので，3 行目のプログラムは構文検査に失敗してしまう．

図 16 の文法をうまく変更しても，Yamazaki らの手法では，図 17 の最後の行の構文検査は実現することはできない．図 17 の最後の行を実現するには，*ColumnList* から *ColumnList ColumnList* が導出できなければならない．しかし，そのようなことのできる文法は曖昧である．*ColumnList* から *ColumnList ColumnList ColumnList* を導出する列として，

$$
\begin{aligned}
&\underline{\mathit{ColumnList}} \\
\longrightarrow^* &\underline{\mathit{ColumnList}}\ \mathit{ColumnList} \\
\longrightarrow^* &\mathit{ColumnList}\ \mathit{ColumnList}\ \mathit{ColumnList}\ \underline{\mathit{ColumnList}} \\
\longrightarrow^* &\mathit{ColumnList}\ \underline{\mathit{ColumnList}} \\
\longrightarrow^* &\mathit{ColumnList}\ \mathit{ColumnList}\ \mathit{ColumnList}
\end{aligned}
\tag{1}
$$

の 2 種類が考えられてしまうからである．ただし，式 (1) において下線部は直後で導出により置き換えられる場所を示している．

4.2.2 解決法

図 17 の最終行の構文検査は，4.1.2 節で示した解決策と同様の手法で可能になる．Yamazaki らの手法における新たな非終端記号を追加するときに文法の変形を工夫する．

文法の変形としては図 18 のようにおこなう．ϵ

$$
\begin{aligned}
\mathit{Query} &::= \mathit{Select}\ \mathit{From}\ \mathit{Where} \\
\mathit{Select} &::= \texttt{select()}\ \mathit{ColumnList} \\
\mathit{ColumnList} &::= \texttt{column(string)}^* \\
\mathit{From} &::= \texttt{from()}\ \texttt{table(string)} \\
\mathit{Where} &::= \texttt{where()}\ \texttt{column(string)}\ \mathit{Op}\ \texttt{num(int)} \\
\mathit{Op} &::= \texttt{le()} \mid \texttt{ge()} \mid \texttt{eq()}
\end{aligned}
$$

図 16: 非終端記号 *ColumnList* を追加した文法の例

```
ColumnList columns1 = column("PRICE") + column(
    "PRODUCT_ID");
ColumnList columns2 = column("QUANTITY") +
    column("NAME");
ColumnList columns3 = columns1 + columns2;
```

図 17: *ColumnList* で型付けされたトークン列の断片同士を結合するプログラム例

は空列を表している．この文法では *ColumnList* は `column(string)` と $\widehat{\texttt{ColumnList}}$ がいくつか並んだものとなっている．$\widehat{\texttt{ColumnList}}$ は *ColumnList* から導出されると型付けされたトークン列である．そのため，図 17 のように `ColumnList` と型付けされた変数を複数ならべたものも *ColumnList* から導出されると判定できるようになる．

図 18 の文法では *ColumnList* の代わりに $\widehat{\texttt{ColumnList}}$ を用いることで，導出 $\mathit{ColumnList} \longrightarrow^* \mathit{ColumnList}\ \mathit{ColumnList}$ が生じない．そのため，文法が曖昧になることを避けることができている．

同様の変形は $A \longrightarrow X^*$ の形の文法があったときに適用可能である．$A \longrightarrow X^*$ の生成規則を $A \longrightarrow X\ A \mid \widehat{\texttt{A}}\ A \mid \epsilon$ と変形する．この変形によって，A から導出されるトークン列同士を結合させたときに A から導出されたものとして型が付く．変形前の文法が曖昧でなく，かつ A から導出されるものが X^* のみであるなら，変形後の文法も曖昧でない．

5 実装

本研究では，3 章と 4 章で提案した演算子でトークンをつなぐ fluent API および，結合を柔軟にするための文法変形の一部を利用できる fluent API 生成器

$$
\begin{aligned}
\mathit{Query} &::= \mathit{Select}\ \mathit{From}\ \mathit{Where} \\
\mathit{Select} &::= \texttt{select()}\ \mathit{ColumnList} \\
&\mid \widehat{\texttt{Select}} \\
\mathit{ColumnList} &::= \texttt{column(string)}\ \mathit{ColumnList} \\
&\mid \widehat{\texttt{ColumnList}}\ \mathit{ColumnList} \\
&\mid \epsilon \\
\mathit{From} &::= \texttt{from()}\ \texttt{table(string)} \\
&\mid \widehat{\texttt{From}} \\
\mathit{Where} &::= \texttt{where()}\ \texttt{column(string)} \\
&\quad \mathit{Op}\ \texttt{num(int)} \\
&\mid \widehat{\texttt{Where}} \\
\mathit{Op} &::= \texttt{le()} \mid \texttt{ge()} \mid \texttt{eq()} \\
&\mid \widehat{\texttt{Op}}
\end{aligned}
$$

図 18: 図 17 のプログラムの構文検査を可能にする文法

の実装を行っている．ターゲット言語としては，演算子オーバーロードや高度な型の操作が可能である言語である C++を採用した．

図 19 は fluent API 生成器への入力例である．文脈自由文法を記述して入力を行う．非終端記号は大文字始まりの変数名，終端記号は小文字始まりの名前になっている．終端記号は引数を取ることができ，() の中に型を記述する．

入力された文脈自由文法に対して，まず文法の変形を行う．2.3 節で示した Yamazaki らの手法と同じ変形に加えて，本研究特有の変形として $X \longrightarrow A^*$ の形の文法があったときに，4.2 節で示した文法の変形を行う．4.1 節で示した文法変形については未実装であり，実装することは今後の目標である．

```
start: Query;
Query => Select From Where;
Select => select() ColumnList;
ColumnList => column(string)*;
From => from() table(string);
Where => where() column(string) Op num(int);
Op => le() | ge() | eq();
```

図 19: Fluent API 生成器への入力例

文法を変形したあとに構文検査のために必要な型や関数定義の生成を行う．構文検査については C++を用いた先行研究 [11] と同様に型システムを用いて LR オートマトンをシミュレートして構文検査を行う．

今回実装したシステムにおいて，構文検査が行われるタイミングは，多くの先行研究とは異なる．先行研究ではメソッド呼び出しが行われるたびに，型検査によってその呼出しが構文仕様に沿った呼び出しかが検査される．一方で今回実装したシステムでは，演算子で結ばれた時点ではその構文的な正しさは判定されない．演算子による返り値の型としては，トークン列を構成しているトークンがエンコードされたもの返す．例えば，`select() + column("PRICE")` の返り値の型は，`TokenList<select_terminal, column_terminal>`のような型となる．構文検査が行われるのは明示的にコンストラクタが呼び出されるか，トークン列が変数に束縛されるなど暗黙の型変換が行われるタイミングである．例えば，図 9 のプログラムでは変数 `query` にトークン列が束縛されたときに発生する型 `Query` への暗黙の型変換によって構文検査が行われる．構文検査では，トークン列の型の情報からそのトークン列が変換先の非終端記号から導出可能であるかを判定する．

6 関連研究

Fluent API は Fowler によって fluent interface として提案され [6]，その構文検査については多くの研究が存在する．

Yamazaki ら [11] は LR 文法の文法仕様に対する構文検査を Scala, C++, Haskell の型システム上で実現する手法を提案した．この研究では，実際に fluent API 生成器の作成を行っている．Gil と Roth [4] は，決定性文脈自由文法の文法仕様に対する構文検査手法を Java の型システム上で実現する手法を示した．この手法では，決定性プッシュダウンオートマトンを型でシミュレートする．決定性文脈自由言語は LR(k) 文法と等価であるが，この研究で作成された fluent API 生成器は LL(1) 文法のみを対象としている．この 2 つの研究では flat-chaining style のみを対象としている．

Roth ら [9] は，決定性文脈自由言語の文法仕様に対して Standard ML の型システムを用いて構文検査を行う手法を示している．この研究では sub-chainining style を扱う手法も示しているが，トークン列の断片を直接構文検査するものではない．

トークン列の断片の構文検査を対象にした sub-chainining style の研究は次のようなものがある．Nakamaru ら [7] は文脈自由文法の一部について sub-chainining style を実現する手法を示し，fluent API 生成器の実装を行った．Yamazaki ら [10] 文法のこの研究では，fluent API 生成器の実装までは行っていない．

これまでの fluent API に対する構文検査の研究では，いずれも曖昧な文法を扱っていない．本研究では曖昧な文法を扱っている．

構文解析において，曖昧な文法を扱う研究は多くなされている．パーサージェネレーターである Yacc [5] では演算子に優先順位をつけることによって曖昧性の回避を行っている．構文解析の際に，演算子の優先順位に応じて複数ある構文木の候補から適切なものを選択する [2]．Aasa らは文法に演算子の優先順位を導入することが，文法の変形のみによって可能であることを示している [1]．近年では，曖昧な文が与えられたときにユーザーに曖昧でなくなるようにプログラムを書き直させるアプローチが提案されている [8]．

参 考 文 献

[1] Aasa, A.: Precedences in specifications and implementations of programming languages, *Theoretical Computer Science*, Vol. 142, No. 1(1995), pp. 3–26.

[2] Aho, A. V., Johnson, S. C., and Ullman,

J. D.: Deterministic parsing of ambiguous grammars, *Communications of the ACM*, Vol. 18, No. 8(1975), pp. 441–452.

[3] Alfred, V. A., Monica, S. L., and Jeffrey, D. U.: *Compilers principles, techniques & tools*, pearson Education, 2007.

[4] Gil, Y. and Roth, O.: Fling-A fluent API generator, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019.

[5] Johnson, S. C. et al.: *Yacc: Yet another compiler-compiler*, Vol. 32, Bell Laboratories Murray Hill, NJ, 1975.

[6] Martin, F.: Fluent Interface, `https://www.martinfowler.com/bliki/FluentInterface.html`. (Accessed on 07/29/2024).

[7] Nakamaru, T., Ichikawa, K., Yamazaki, T., and Chiba, S.: Silverchain: a fluent API generator, *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2017, pp. 199–211.

[8] Palmkvist, V., Castegren, E., Haller, P., and Broman, D.: Statically Resolvable Ambiguity, *Proceedings of the ACM on Programming Languages*, Vol. 7, No. POPL(2023), pp. 1686–1712.

[9] Roth, O. and Gil, Y.: Fluent APIs in Functional Languages, *Proceedings of the ACM on Programming Languages*, Vol. 7, No. OOPSLA1(2023), pp. 876–901.

[10] Yamazaki, T., Nakamaru, T., and Chiba, S.: Yet Another Generating Method of Fluent Interfaces Supporting Flat-and Sub-Chaining Styles, *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, 2022, pp. 249–259.

[11] Yamazaki, T., Nakamaru, T., Ichikawa, K., and Chiba, S.: Generating a fluent API with syntax checking from an LR grammar, *Proceedings of the ACM on Programming Languages*, Vol. 3, No. OOPSLA(2019), pp. 1–24.