# 内部DSLの有効範囲を制御するための言語機構の提案

市川 和央[1], 千葉 滋[2]

東京大学情報理工学系研究科

[1] ichikawa@csg.ci.i.u-tokyo.ac.jp

[2] chiba@acm.org

**概 要** 我々は、内部ドメイン専用言語 (内部 DSL) の文脈情報を文脈オブジェクトと呼ぶオブジェクトとして表現し、それを暗黙的にやりとりする、内部 DSL の文脈を表す言語機構を提案する。本提案機構を用いることで、内部 DSL の途中状態をユーザの変数を介さずにそれ以降の内部 DSL の式に渡すことができるため、簡潔で安全性の高い内部 DSL を実装可能となる。文脈オブジェクトは内部 DSL の式により登録解除することもでき、その文脈オブジェクトを利用した内部 DSL は登録から登録解除の間でのみ利用できる。言い換えると、本提案機構は内部 DSL のスコープとその内部でのローカルな状態を表現するものである。我々はこのような言語機構をコンパイル型の静的型付き言語上に実装できるよう設計した。現在、我々は前身研究である ProteaJ に提案した言語機構を追加したプログラミング言語 ProteaJ2 の実装を行っている。

## 1 はじめに

プログラムは簡潔であることが望ましい。これは単にプログラムを記述するための労力が減るだけではなく、機械的に推論できる部分をコンパイラ等に任せることで、人為的なバグの減少にもつながる。実際、多くのプログラミング言語はプログラムの一部の記述を省略してプログラムを簡潔にする機能を持っている。例えば、C# や Scala [7] は implicit conversion と呼ばれる自動型変換機能を持つ。他にも、C++ を始めとする多くの言語はデフォルト引数と呼ばれる引数の省略機能を持ち、Scala もそれに似た implicit parameter と呼ばれる機能を持つ。このような省略機能はプログラムの動きをわかりづらくしてしまうという批判もあるが、実際の動作を隠蔽して宣言的にプログラムを記述する上では非常に有用である。

内部ドメイン専用言語 (内部 DSL) [5] はプログラミング言語の省略機能や文法拡張機能を活用して実装されたライブラリで、対象となるドメインのプログラムを簡潔に記述することができる。内部 DSL は独立した言語ではないため文法に制限がかかるが、他の内部 DSL やそれを実装した汎用言語 (ホスト言語) と容易に連携できるという利点を持つ。我々は以前から内部 DSL を支援するための研究を行っており [6]、実際に ProteaJ と呼ぶ Java を拡張した言語を開発している。

本研究は ProteaJ を拡張してより自由度の高い内部 DSL の設計を可能とすることを目標としている。ProteaJ は *protean operators* と呼ぶユーザ定義演算子によって内部 DSL を表現する言語で、Parsing Expression Grammar (PEG) [4] で表現可能な任意の内部 DSL の文法を実現することができる。しかし、protean operators はあくまで演算子であるため、表現できるのは式レベルの内部 DSL である。複数の文によって構成される内部 DSL を実現するには、ユーザの変数を介してそれらが共通して内部で用いる値を共有する必要がある。これはプログラムを冗長にするだけでなく、カプセル化の観点からも望ましくない。なお、この問題は ProteaJ に限ったものではなく、内部 DSL をホスト言語の式により実装した場合に発生するものである。本研究ではこのような問題を解決し、より簡潔かつ安全な、複数の文からなる内部 DSL を実装可能にすることを目指す。

我々は、文をまたいだ文脈情報を 文脈オブジェクト と呼ぶオブジェクトとして、それ以降の内部 DSL の式に暗黙的に渡す機構を提案する。こうすることで、カプセル化すべき状態をユーザの変数を介さずに、続けて記述されている内部 DSL の式に渡すことができる。文脈オブジェクトは内部 DSL の式により登録解除することもでき、その文脈オブジェクトを利用した内部 DSL は登録から登録解除の間でのみ利用できる。言い換えると、本提案機構は内部 DSL のスコープとその内部でのローカルな状態を表現するものである。我々は、このような言語機構をコンパイル型の静的型付き言語上に実装できるよう設計した。現在、我々は ProteaJ に提案した言語機構を追加した言語 ProteaJ2 の実装を行っている。

図 1 は SQL を利用したプログラムの例で、A は Java、B は ProteaJ、C は ProteaJ2 で記述されている。ProteaJ は式部分の文法をユーザ定義できるため、例えば Java では

```
driver.connect("jdbc:postgresql:database", new Properties())
```

となっていた部分は、ProteaJ では

```
connect jdbc:postgresql:database by driver
```

という式となっている。ここで、ProteaJ ではリテラルの文法も演算子で表現できるため、接続するデータベースは文字列ではなくユーザ定義リテラルで渡されている。この内部 DSL では、`Driver` や `Connection` といった SQL 内部でしか用いられないオブジェクトをユーザに渡しており、カプセル化が十分でないといえる。
ProteaJ2 では、

```
use org.postgresql.Driver
```

が `Driver` を初期化して、それを含む文脈オブジェクトを呼び出し文脈に登録し、

```
connect jdbc:postgresql:database
```

がその文脈オブジェクトを利用するため、`driver` 変数が不要となる。また、ユーザから見ると `Driver` や `Connection` オブジェクトは隠蔽されており、ユーザが明示的にこれらのオブジェクトを操作している B よりも安全性が高い。

```
// A: Java による記述
Driver driver = (Driver)Class.forName("org.postgresql.Driver").newInstance();
Connection conn = driver.connect("jdbc:postgresql:database", new Properties());
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from students where score > 80");
while (rs.next()) {
  System.out.println(rs.getString("name") + " : " + rs.getInt("score"));
}
conn.close();

// B: ProteaJ による記述
Driver driver = org.postgresql.Driver;
Connection conn = connect jdbc:postgresql:database by driver;
ResultSet rs = select * from students where score > 80 by conn;
while (rs.next()) {
  println rs[name]:str + " : " + rs[score]:int;
}
close conn;

// C: ProteaJ2 による記述
use org.postgresql.Driver;
connect jdbc:postgresql:database;
select * from students where score > 80;
while (each result) {
  println name:str + " : " + score:int;
}
disconnect;
```

**図 1.** SQL 用内部 DSL を用いたプログラム

本論文の残りの部分の構成は以下の通りである。2 節では、前身研究となる ProteaJ について簡単に解説した後、状態を持つ内部 DSL を実現する際の問題点について述べる。3 節では、我々の提案する言語機構を現在実装中の言語 ProteaJ2 のコード例を交えて解説する。4 節では提案機構の利用例をいくつか紹介する。5 節では関連研究を紹介し、6 節で論文全体のまとめを述べる。

## 2 状態を持つ内部 DSL

本研究は、状態を持つ内部 DSL をデザインする際に、文脈を表現する言語機構が有用であることを示すものである。そのために、まず本研究の前身研究である ProteaJ を簡単に紹介し、その後、ProteaJ 上での状態を持つ内部 DSL の実装方法とその問題点を述べる。なお、ここで述べる問題点は ProteaJ に限らず、内部 DSL をホスト言語の式で表現する場合に発生する。そのため、提案手法は ProteaJ 以外の言語に適用した場合も有用であると考えられる。

## 2.1 ProteaJ

ProteaJ は本研究の前身研究である [6] で実装されたプログラミング言語で、*protean operators* と呼ぶユーザ定義演算子により内部 DSL を表現する。protean operators は名前部分と引数部分の組み合わせで表現可能な演算子で、引数の型と返り値の型でオーバーロードされるという特徴を持つ。名前部分は単純な単項・二項演算子であれば演算子名に当たる部分で、例えば足し算演算子であれば + が名前部分となる。以下では名前部分はわかりやすさのためにダブルクォートで囲って表示するものとする。引数部分は演算子のオペランドに当たる部分で、以下では引数部分を `_` と記述する。例えば、足し算演算子は `_ "+" _` のような文法を持つ演算子である。名前部分と引数部分の並び方や個数は任意で、例えば `"read" "line"` のような演算子や `_ _ _` のような演算子も protean operators に含まれる。

protean operators は引数の型と返り値の型の両方でオーバーロードされるという特徴を持つが、これは言い換えると、引数の型と返り値の型の両方が合っているときにのみ、その演算子が利用されるということである。そのため、似た文法を持つ演算子を同時に利用しても誤った解析を起こしにくいという利点を持つ。複数の内部 DSL を同時に利用できるという性質は、内部 DSL の *composability* と呼ばれる重要な性質である [8]。

protean operators のもう一つの特徴として、演算子間の優先順序がある。protean operators には二種類の優先順序 operator precedence と parsing precedence が存在する。operator precedence は足し算演算子 `_ "+" _` と掛け算演算子 `_ "*" _` の順序関係のような、いわゆる演算子優先順序である。それに対して parsing precedence は演算子優先順序が同じもの同士、例えば足し算演算子 `_ "+" _` と引き算演算子 `_ "-" _` の順序関係である。この優先順序は、構文解析の際にどちらを先に利用するかを表している。足し算演算子 `_ "+" _` と引き算演算子 `_ "-" _` の場合にはどちらを優先しても問題はないが、例えば、SQL の select-from 文を表現する `"select" _ "from" _` と同じく select-from-where 文を表現する `"select" _ "from" _ "where" _` のような演算子が存在した場合には、後者を優先しないと後者は利用されなくなってしまう。

ProteaJ ではこのような protean operators をユーザが自由に定義して利用することができる。protean operators を組み合わせることで、PEG で表現可能な任意の内部 DSL の文法を持つ式を表現できる。また、protean operators は字句レベルの文法も表現することができ、ユーザ定義リテラルを実装することも可能である。ただし、括弧及び中括弧の対応は崩すことはできず (演算子の一部として利用することは可能)、引用符 (')、二重引用符 (")、カンマ (,)、及びコメント記号 (//, /*, */) は演算子に利用できないという制限を持つ。

## 2.2 ProteaJ による状態を持つ内部 DSL の実装

図 1 の例は複雑であるため、ここではもう少し簡単な例を用いて説明を行う。図 2 はファイルから一行ずつ読み込み、それを `Interpreter` オブジェクトの `interpret` メソッ

ドによって解釈する Java プログラムである。ファイル終端まで達するか interpret メソッドの結果がエラーとなったとき、結果を出力して終了する。

図 3 はファイル読み込みのための内部 DSL の ProteaJ における実装例である。ここでは簡単のため、演算子間の優先順序の指定や例外処理の記述を省略している。dsl から始まるブロックが内部 DSL を定義している部分で、ここでは四つの演算子を定義している。例えば、

```
Reader "open" _ (File file) { return new Reader(file); }
```

は Reader 型の返り値を持つ演算子 "open" _ を定義しており、その引数部分の型は File である。この演算子が呼ばれると中括弧内 (演算子ボディ) の処理が実行される。dsl に続く class 定義はこの内部 DSL の状態と処理を格納したクラスの定義である。この内部 DSL を用いると、図 2 のプログラムは図 4 のように記述できる。

図 4 のプログラムの問題点は、ユーザが内部 DSL の状態を格納した Reader オブジェクト reader を扱っている点である。これは以下の二つの問題を引き起こす:

- プログラムが冗長となる
  各内部 DSL の式に reader を渡す必要があり、reader が何度も繰り返し出現する。
- バグの要因となりうる
  close reader の後に read line by reader を呼ぶような、誤った記述が可能。

ユーザが reader を明示的に扱わなくて良いようにする簡単な方法は、静的変数を用いることである。図 5 は静的変数を用いたファイル読み込み内部 DSL の実装例である。ここでは reader が静的変数であり、内部 DSL の状態を保持する Reader オブジェクトを格納する。演算子 "open" _ によってファイルを開いた時、reader に新しく作られた Reader オブジェクトが格納され、それ以降で演算子 "eof" や演算子 "read" "line" が呼ばれると、この Reader オブジェクトを利用して計算が行われる。この内部 DSL を用いると、図 2 のプログラムは図 6 のように記述できる。

図 6 のプログラムは内部 DSL の状態を格納した Reader オブジェクトをユーザが明示的に扱っていないため、図 4 のプログラムよりも簡潔になっている。しかし、この解決法

```
Interpreter interpreter = new Interpreter();
BufferedReader reader = new BufferedReader(new FileReader(file));
while (true) {
  String line = reader.readLine();
  if (line == null) break;
  else {
    Result r = interpreter.interpret(line);
    if (r.isError()) break;
  }
}
reader.close();
interpreter.printResult();
```

**図 2.** 一行ずつ読み込み解釈するプログラム

```
dsl ReaderDSL {
  Reader "open" _ (File file) { return new Reader(file); }
  boolean _ "reaches" "eof" (Reader reader) { return reader.eof(); }
  String "read" "line" "by" _ (Reader reader) { return reader.readLine(); }
  void "close" _ (Reader reader) { reader.close(); }
}
class Reader {
  Reader (File file) {
    this.reader = new BufferedReader(new FileReader(file));
    this.nextLine = reader.readLine();
  }
  String readLine() {
    String line = nextLine;
    nextLine = reader.readLine();
    return line;
  }
  boolean eof() { return nextLine == null; }
  void close() { return reader.close(); }
  private BufferedReader reader;
  private String nextLine;
}
```

**図 3.** ProteaJ によるファイル読み込み内部 DSL の実装例

```
Interpreter interpreter = new Interpreter();
Reader reader = open file;
while (! reader reaches eof) {
  Result r = interpreter.interpret(read line by reader);
  if (r.isError()) break;
}
close reader;
interpreter.printResult();
```

**図 4.** 図 3 の内部 DSL の利用例

では内部 DSL の状態を同時に一つしか持つことができないという問題がある。これは例えば、`interpreter.interpret` が内部で図 5 の内部 DSL を利用していた場合、内部状態を表す `reader` が途中で変わってしまい、`read line` の結果が意図したファイルとは異なるファイルから読み込んだものになってしまう。

そこで我々は、内部 DSL の状態を内部 DSL の文脈ごとに保持すればよいのではないかと考えた。`open file` で生成された `Reader` オブジェクトが、同じ文脈上の内部 DSL の式に暗黙的に渡されるような仕組みが存在すれば、図 6 のような記述が可能かつ安全な内部 DSL が実現できる。問題は、“内部 DSL の文脈” をどう定義し、表現するかという点である。我々は内部 DSL の文脈をコンパイル型の静的型付き言語上に実装できるよう設計した。次節では、内部 DSL の文脈を我々がどう定義し、どのように文脈を表現する言語機能をデザインしたかを述べる。

```
dsl ReaderDSL {
  static Reader reader = null;
  void "open" _ (File file) { reader = new Reader(file); }
  boolean "eof" () { if (reader != null) return reader.eof(); else return true; }
  String "read" "line" ()
    { if (reader != null) return reader.readLine(); else return null; }
  void "close" () { if (reader != null) reader.close(); reader = null; }
}
```

**図 5.** 静的変数を用いたファイル読み込み内部 DSL の実装例

```
Interpreter interpreter = new Interpreter();
open file;
while (! eof) {
  Result r = interpreter.interpret(read line);
  if (r.isError()) break;
}
close;
interpreter.printResult();
```

**図 6.** 図 5 の内部 DSL の利用例

# 3 内部 DSL の文脈を表現する言語機構

我々は、内部 DSL の文脈情報を 文脈オブジェクト と呼ぶオブジェクトとして表現し、それを暗黙的にやりとりする、内部 DSL の文脈を表す言語機構を提案する。ここでは内部 DSL の文脈を、静的なスコープ内で伝播する暗黙的な情報として定義する。提案する言語機構は、内部 DSL の式によって文脈に文脈オブジェクトを追加・削除することができるものである。例えば、図 7 のコードを考える。ここでは、`open file` が `Reader` 文脈オブジェクトを文脈に追加し、`close` がそれを文脈から削除している。そのため、その間では `Reader` 文脈オブジェクトを見つけられるため、`read line` を実行することができる。しかし、文脈は静的なスコープ内でのみ伝播するので、関数 `read_func` を呼び出した先では `read line` は `Reader` 文脈オブジェクトを見つけられず、エラーとなる。これはつまり、文脈オブジェクトは内部 DSL のスコープとその内部でのローカルな状態を表現しているとも言える。

この言語機構は Scala などの言語が持つ implicit parameter とよく似ている。しかし、implicit parameter は暗黙的な値が自明な値でない場合はユーザが登録する必要があり、ユーザが明示的に内部 DSL の状態を表すオブジェクトを扱う必要があるという問題は解決できていない。

我々は提案する言語機構をコンパイル型の静的型付き言語上に実装できるよう設計し、ProteaJ の提供する文法拡張機能を合わせて用いることができるよう、新しいプログラミング言語 ProteaJ2 をデザインした。ProteaJ2 のコンパイラは現在実装中である。本節の残りの部分では、提案する言語機構の ProteaJ2 上でのデザインを紹介する。

```
void open_read_close() {
  open file;        // Reader 文脈オブジェクトを文脈に追加
  read line;        // 文脈から Reader 文脈オブジェクトを利用
  read_func();
  close;            // Reader 文脈オブジェクトを文脈から削除
}
void read_func() {
  read line;        // エラー: 文脈に Reader 文脈オブジェクトは存在しない
}
```

**図 7.** 文脈の伝播と文脈オブジェクト

```
dsl ReaderDSL {
  void "open" _ (File file) activates Reader {
    activate new Reader(file);
  }
  context Reader {
    Reader (File file) {
      this.reader = new BufferedReader(new FileReader(file));
      this.nextLine = reader.readLine();
    }
    boolean "read" "line" () {
      String line = nextLine;
      nextLine = reader.readLine();
      return line;
    }
    boolean "eof" () { return nextLine == null; }
    void "close" () deactivates Reader { return reader.close(); }
    private BufferedReader reader;
    private String nextLine;
  }
}
```

**図 8.** ProteaJ2 によるファイル読み込み内部 DSL の実装例

### 3.1 ProteaJ2

ProteaJ2 は ProteaJ を拡張した言語で、文脈クラス、activates 節、deactivates 節、及び activate 文 を提供することで提案する言語機構を実現する。文脈クラスは通常の Java のクラスと似たモジュール機構で、コンストラクタと演算子とフィールドを持つ。文脈クラスのインスタンスが文脈オブジェクトとなる。activates 節及び deactivates 節は引数として文脈オブジェクトの型 (文脈クラス名) を取り、関数もしくは演算子が指定した型の文脈オブジェクトを文脈に追加または削除することを示す。activate 文は実際に文脈に追加する文脈オブジェクトを指定する文である。activates 節と activate 文の関係は Java の throws 節と throw 文の関係と似ている。

図 8 は ProteaJ2 におけるファイル読み込みのための内部 DSL の実装例である。dsl から始まるブロックが内部 DSL を定義している部分で、ここでは一つの演算子 "open" _ と一つの文脈クラス Reader を定義している。演算子 "open" _ の activates Reader は、

```
Interpreter interpreter = new Interpreter();
open file;                          // Reader 文脈オブジェクト (r1 とする) を文脈に追加
while (! eof) {                     // eof は r1.eof() のように振る舞う
  Result r = interpreter.interpret(read line);
                                    // read line は r1.readLine() のように振る舞う
  if (r.isError()) break;
}
close;                              // close は r1.close() のように振る舞い、r1 を文脈から削除
interpreter.printResult();
```

**図 9.** 図 8 の内部 DSL の利用例

この演算子が呼ばれた場所の文脈に Reader 型の文脈オブジェクトを追加することを示している。また、この演算子の演算子ボディ(中括弧内) では、new Reader(file) によって Reader 文脈オブジェクトを生成し、activate 文によってそれを実際に文脈に追加している。context から始まるブロックは文脈クラスの定義部分であり、ここでは一つのコンストラクタと三つの演算子と二つのフィールドを持つ文脈クラス Reader を定義している。文脈クラス内の演算子はその文脈クラスに含まれるフィールドにアクセスすることができる。例えば、演算子 "read" "line" は reader フィールドの readLine メソッドを呼び出して nextLine フィールドを更新している。演算子 "close" の deactivates Reader は、Reader 文脈クラスをこの演算子を呼び出した文脈から削除することを示している。

文脈クラスに含まれる演算子は、そのインスタンスである文脈オブジェクトが含まれる文脈でのみ利用することができる。例えば、図 8 の演算子 "read" "line" は演算子 "open" _ を呼んでから演算子 "close" を呼ぶまでの間でのみ利用可能となる。その間での演算子 "read" "line" の呼び出しは、演算子 "open" _ が追加した文脈オブジェクトのインスタンス上で計算される。言い換えると、文脈クラスの演算子呼び出しは文脈に含まれる文脈オブジェクトに対するインスタンスメソッド呼び出しのように振る舞う。implicit parameter と異なるデザインを選んだのは、有効な演算子の数を少なくして構文解析を簡単にするためである。

図 8 の内部 DSL を用いると、図 9 のようなプログラムが記述できる。open file を実行すると Reader 文脈オブジェクトが生成され、文脈に追加される。追加された文脈オブジェクトを r1 とする。これ以降では Reader 文脈クラスに含まれる演算子を利用することが可能となり、それらの演算子呼び出しは、r1 をレシーバとしたインスタンスメソッド呼び出しのように振る舞う。例えば、eof は r1.eof() のようなメソッド呼び出しのように振る舞う。close が実行されると、r1 は文脈から削除され、Reader 文脈クラスに含まれる演算子は利用不可能となる。

### 3.2 コンパイラ

文脈オブジェクトは実行時情報を保持するオブジェクトだが、文脈に含まれる文脈オブジェクトの型や演算子呼び出しで利用される文脈オブジェクトを静的に決定できるよう、

我々は ProteaJ2 を設計した。そのため、ProteaJ2 は実行時の構文解析や意味解析を必要としないコンパイル型の静的型付き言語として実装できる。

そのためのもっとも重要な仕組みが activates 節及び deactivates 節である。これらの情報を利用することで、コンパイラは現在の文脈にどの文脈オブジェクトが含まれているかを知ることができる。deactivates 節により削除される文脈オブジェクトは、型が一致する文脈オブジェクトのうち、最後に追加されたものである。型が合う文脈オブジェクトが存在しなかった場合は何も起こらない。deactivate 文が存在しないのは、文脈から削除される文脈オブジェクトをコンパイル時に一意に決めるためである。

文脈オブジェクトは同一の型を持つものを複数文脈に追加することも可能だが、文脈クラスの演算子呼び出しで用いられる文脈オブジェクトは、その文脈クラスのインスタンスのうち、最後に文脈に追加されたものとなる。この特徴により、コンパイラは演算子呼び出しで用いられる文脈オブジェクトをコンパイル時に一意に決めることができる。

また、静的な解析を可能とするために、if 文のような制御構造に対する制限が存在する。どのパスを通った場合でも、合流する時点で文脈に含まれる文脈オブジェクトの型が、追加された順序を含めて一致していなければならない。これも、コンパイラが文脈に含まれる文脈オブジェクトを知るための制限である。

以上のように ProteaJ2 を設計したことで、意味解析の結果を次の解析に利用しつつ再帰下降構文解析を行うことで、構文解析をすることが可能となっている。現在構文解析をしている部分よりも前の部分を意味解析して、activates 節及び deactivates 節を持つ関数もしくは演算子の呼び出しを見つけることで、文脈に含まれる文脈オブジェクトが判定できる。現在構文解析をしている部分で利用可能な演算子を知ることができるため、ProteaJ と同様の構文解析手法によって高速に構文解析を行うことが可能となる。

## 3.3 発展的機能

### 3.3.1 requires 節

本提案では、内部 DSL の文脈を静的なスコープ内で伝播する暗黙的な情報として定義したが、文脈情報を動的に伝播させて利用したい場合は多く存在すると考えられる。もし、図 7 の例で内部 DSL の文脈が動的に伝播するならば、関数 read_func を呼び出した先でも open file によって作られた文脈オブジェクトが有効となるため、read line も正しく実行される。しかし、read line を解釈するためには文脈情報が必要であるため、内部 DSL の文脈が動的に伝播するものとするのであれば、実行時に構文解析を行わなければならなくなる。これはパフォーマンスに大きな悪影響を及ぼしてしまう。

我々は、requires 節を提供することでこれをサポートしようと考えている。requires 節は引数として文脈オブジェクトの型を取り、関数もしくは演算子が指定した型の文脈オブジェクトを含む文脈で呼び出さなければならないことを示す。指定された型の文脈オブジェクトは、その関数もしくは演算子の内部に伝播する。これは以下のようにして用いる。

```
void read_func() requires Reader {
  read line;
```

```
}
```

こうすることで、関数 read_func は Reader 文脈オブジェクトを含む文脈からしか呼び出せなくなり、read line は正しく解釈できるようになる。

### 3.3.2 ホスト言語のスコープと文脈オブジェクト

あるホスト言語のスコープで文脈オブジェクトが追加・削除されたとき、その外側のスコープではその文脈オブジェクトをどのように扱えば良いかは自明ではない。ProteaJ2 ではこれを二つのオプションから選択可能にする。どちらをデフォルトとして採用するかは現在検討中である。例えば、以下の様な場合を考える。

```
{             // ブロック: ホスト言語のスコープ
  open file;  // Reader 文脈オブジェクトを追加
  read line;  // OK
}
read line;    // どうする？
```

このとき、open file を呼んだスコープの外で、Reader 文脈オブジェクトが文脈に含まれるべきかどうかは、その内部 DSL の設計によって異なる。

ProteaJ2 では Reader 文脈クラスの実装時に propagate 修飾子を指定した場合、open file を呼んだスコープの外側でも、Reader 文脈オブジェクトが文脈に含まれる。propagate 修飾子は、ホスト言語のスコープの外まで文脈オブジェクトの追加・削除が伝播することを表している。propagate 修飾子が指定されている場合、ホスト言語の関数もしくは演算子のボディの外側まで、文脈オブジェクトの追加・削除が伝播することがある。その場合、activates 節または deactivates 節を記述して、その関数もしくは演算子を利用した場所の文脈にも文脈オブジェクトの追加・削除が伝播することを示す必要がある。propagate 修飾子が指定された文脈クラスの文脈オブジェクトは、while 文や for 文のようなループ構造の内部で文脈に追加した場合、必ずループ構造の内部で文脈から削除しなければならない。これは、どのパスを通った場合でも文脈に含まれる文脈オブジェクトの型が同じでなければならない、という制限によるものである。

Reader 文脈クラスの実装時に auto-close 修飾子を指定した場合、open file を呼んだスコープの外側では Reader 文脈オブジェクトは文脈に含まれない。auto-close 修飾子は、文脈オブジェクトが追加されたスコープを抜けると、自動的にその文脈オブジェクトが文脈から削除されることを示している。その際に自動的に呼ばれる後処理を指定することも可能で、そのような文脈クラスは次のようにして定義する。

```
auto-close context Reader {
  Reader (File file) { ... }
  ~Reader () { reader.close(); }    // 自動的に呼ばれる後処理
  boolean "read" "line" () { ... }
  boolean "eof" () { ... }
  private BufferedReader reader;
  private String nextLine;
}
```

この後処理は deactivates 節を持つ関数もしくは演算子によって文脈から削除された場合には呼び出されない。後処理によって文脈オブジェクトを追加することも可能で、そのような文脈オブジェクトはスコープを抜けた外のスコープに追加される。auto-close 修飾子が指定された文脈クラスの文脈オブジェクトは、それが追加されたスコープよりも内側のスコープで文脈から削除することはできない。例えば、以下の様なプログラムはエラーとなる。

```
open file;    // Reader 文脈オブジェクトを追加
{             // ブロック: ホスト言語のスコープ
  read line;  // OK
  close;      // エラー: 追加されたスコープでのみ削除可能
}
```

これは、propagate 修飾子の場合と異なり、文脈オブジェクトの削除が伝播しないためである。削除のみ伝搬するというのは一貫性に欠けるため、我々はこのようなデザインを選択した。

なお、現在のところ例外によってスコープを脱出した場合については考慮していない。これは、例外による脱出を含めて考えると、どのパスを通った場合でも文脈に含まれる文脈オブジェクトの型が同じでなければならない、という制限をチェックするのが困難になると考えられるためである。

# 4 ケーススタディ

本節では、提案した言語機構を用いた内部 DSL の具体例を紹介する。

## 4.1 アプリケーションの設定

提案した言語機構は内部 DSL の状態によって有効な演算子を切り替えられるため、ビルダパターンのような記述を行うのに適している。図 10 は最初に設定を行い、設定終了後は設定された通りに動作するログ出力 API を提供する内部 DSL である。演算子 "configure" を呼ぶことで、文脈オブジェクト Configuring が文脈に追加され、設定を記述するための演算子が利用可能となる。Configuring 文脈クラスには auto-close 修飾子が付けられているため、演算子 "configure" を呼んだスコープを抜けたところで、文脈オブジェクト Configuring は自動的に文脈から削除される。このとき、後処理 Configuring が呼ばれ、文脈オブジェクト Configured が文脈に追加される。そのため、演算子 "configure" を呼んだスコープの外側のスコープでは、debug や error などの各種ログ出力 API が利用可能となる。

図 11 は図 10 の内部 DSL を利用したプログラムの例である。このプログラムは図 10 の内部 DSL 以外にも、File 及び LogLevel のリテラルを表現する内部 DSL を併用しているが、これらの内部 DSL は文脈情報を用いていないため、説明は割愛する。図 10 の内部 DSL を利用して、ログの設定を行っているのが関数 setting である。関数 setting のボ

```
dsl AppConfigDSL {
  void "configure" () activates Configuring {
    activate new Configuring();
  }
  auto-close context Configuring {
    Configuring() {
      logFile = new File("./log.txt");
      logLevel = LogLevel.warn;
    }
    ~Configuring() activates Configured {
      activate new Configured(logFile, logLevel);
    }
    void "log" "file" "=" _ (File file) { logFile = file; }
    void "log" "level" "=" _ (LogLevel level) { logLevel = level; }
    File logFile;
    LogLevel logLevel;
  }
  auto-close context Configured {
    Configured(File logFile, LogLevel logLevel) {
      logger = new Logger(logFile, logLevel);
    }
    ~Configured() { logger.close(); }
    void "debug" "(" _ ")" (String msg) { logger.debug(msg); }
    void "info" "(" _ ")" (String msg) { logger.info(msg); }
    void "warn" "(" _ ")" (String msg) { logger.warn(msg); }
    void "error" "(" _ ")" (String msg) { logger.error(msg); }
    Logger logger;
  }
}
```

**図 10.** アプリケーションの設定のための内部 DSL

ディ部分の冒頭で演算子 "configure" を呼んでいるため、文脈オブジェクト Configuring は関数の先頭で文脈に追加され、関数の最後に文脈から削除される。そのため、後処理によって追加される文脈オブジェクト Configured は、この関数の呼び出し側に追加される。関数 setting が activates 節を持つのは、これを表現するためである。関数 main の setting 関数呼び出しは文脈オブジェクト Configured を文脈に追加するので、それ以降で debug や error などの各種ログ出力 API が利用できる。また、関数 main の中で呼ばれている関数 run は、requires 節で文脈オブジェクト Configured を持つ文脈で呼び出すことを要求しているため、関数 run の中でも debug や error などが利用できる。

## 4.2 SQL

図 12 は、本論文の冒頭で紹介した、図 1 の C のプログラムを記述することができる内部 DSL の一部である。簡単に述べると、演算子 "use" _ を使うことで演算子 "connect" _ が利用可能となり、演算子 "connect" _ を使うことで演算子 "select" _ "from" _ "where" _ と演算子 "disconnect" が利用可能となる。更に、演算子 "select" _ "from" _ "where" _ を使うことで演算子 "each" "result" が利用可能となり、演算子 "each"

```
public static void main (String[] args) {
  setting();
  run();
  info("terminated successfully");
}
static void setting() activates Configured {
  configure;
  log file = ./logs/log.txt;
  log level = error;
}
static void run() requires Configured {
  debug("start running");
  ...
}
```

**図 11.** 図 10 の内部 DSL の利用例

"result" を使うことで演算子 _ ":" "str" と演算子 _ ":" "int" が利用可能となる。演算子 _ ":" "str" と演算子 _ ":" "int" は、定義している文脈クラスに auto-close が指定されているため、while 文のスコープを抜けると利用できなくなる。最後に、演算子 "disconnect" を使うことで、演算子 "select" _ "from" _ "where" _ などが利用できなくなる。

# 5 関連研究

## 5.1 型を利用したアプローチ

内部 DSL の状態をユーザから隠蔽する他の手法として、型を利用して内部 DSL のオブジェクトを区別するというものがある。次のようなプログラムがその一例である。

```
Interpreter interpreter = new Interpreter();
Read<Void> interpretLines() {
  return eof().filter(e -> ! e)                  // if (eof()) break;
              .flatMap(e -> readLine())
              .map(interpreter::interpret)       // interpreter.interpret(readLine())
              .filter(r -> ! r.isError())        // if (r.isError()) break;
              .flatMap(r -> interpretLines());   // 再帰でループ
}
open(file).perform(interpretLines());            // open して interpretLines して close
interpreter.printResult
```

ここでは、内部DSLのオブジェクトを Read 型にラップすることで区別している。Read<Void> はファイル読み込みを行って void を返す (つまり何も返さない) ような処理を示す型で、これは処理を表現しているが、実際の実行はこの時点では行われない。Read<Void> 型の interpretLines を open(file).perform に渡すことで、ファイルを開いてから interpretLines で指示された処理を行い、ファイルを閉じるという処理を実際に実行する。map, flatMap, filter はホスト言語で記述した処理を Read 型に対して適用するため

```
dsl SQLDSL {
  void "use" _ (Driver driver) activates UseDriver {
    activate new UseDriver(driver);
  }
  auto-close context UseDriver {
    UseDriver(Driver driver) { this.driver = driver; }
    void "connect" _ (DB db) activates Connecting {
      activate new Connecting(driver.connect(db.path, new Properties()));
    }
    Driver driver;
  }
  propagate context Connecting {
    Connecting(Connection conn) { this.conn = conn; }
    void "select" _ "from" _ "where" _ (Columns col, Tables tbl, Condition cond)
      deactivates QueryResult activates QueryResult
    {
      Statement stmt = conn.createStatement();
      String query = QueryFactory.select_from_where_(col, tbl, cond);
      activate new QueryResult(stmt.executeQuery(query));
    }
    void "disconnect" () deactivates Connecting, QueryResult { conn.close(); }
    Connection conn;
  }
  auto-close context QueryResult {
    QueryResult(ResultSet rs) { this.rs = rs; }
    boolean "each" "result" () activates ResultRow {
      boolean next = rs.next();
      activate new ResultRow(rs);
      return next;
    }
    ResultSet rs;
  }
  auto-close context ResultRow {
    ResultRow(ResultSet rs) { this.rs = rs; }
    String _ ":" "str" (Column c) { return rs.getString(c.name); }
    int _ ":" "int" (Column c) { return rs.getInt(c.name); }
    ResultSet rs;
  }
}
```

**図 12.** SQL のための内部 DSL

に必要な、一種のグルーコードである。このアプローチは高い安全性を持つが、内部 DSL が他の内部 DSL やホスト言語と連携するためにはグルーコードが必要となる。tagless final interpretation [1] はこのようなアプローチの一種で、近年注目されている内部 DSL の表現方法の一つである。本研究は内部 DSL の表面的な文法を扱うものであるため、その内部表現として tagless final interpretation などを採用することで、より安全性や拡張性に優れた内部 DSL を実現できる可能性がある。

### 5.2 オブジェクトインポート

Scala などの言語が持つオブジェクトインポートは、インポートしたオブジェクトのメソッド呼び出しをレシーバを省略して記述できる機能である。これを利用することで、例えば次のようなプログラムが記述できる。ただし、Scala は `break` を持たないため、このコードは擬似コードである。

```
val interpreter = new Interpreter()
val reader = open(file)
import reader._
while (! eof) {                              // eof は reader.eof() の略
  val r = interpreter.interpret(readLine)    // readLine は reader.readLine() の略
  if (r.isError) break
}
close                                        // close は reader.close() の略
interpreter.printResult
```

`reader.` を省略できるため、プログラムが簡潔になっているが、内部 DSL の状態を表現する `Reader` オブジェクトをユーザが明示的に扱う必要がある。

### 5.3 Implicit parameter

Scala などの言語が持つ implicit parameter は自明な引数を省略してプログラムを簡潔にするのに役立つ。これは、引数が省略された時に暗黙的な値を自動的に探して補完する機能である。C++ などが持つデフォルト引数と似ているが、implicit parameter の暗黙的な値はそれが使われた場所の静的なスコープから検索される。そのため、文脈によって異なる値を暗黙的な値として利用することも可能である。しかしその場合はユーザがその暗黙的な値を登録しなければならない。次のコード片は implicit parameter を利用した内部 DSL で記述されたプログラム (擬似コード) である。

```
val interpreter = new Interpreter()
implicit val reader = open(file)
while (! eof) {                              // eof は eof(reader) の略
  val r = interpreter.interpret(readLine)    // readLine は readLine(reader) の略
  if (r.isError) break
}
close                                        // close は close(reader) の略
interpreter.printResult
```

引数の `reader` が省略され、プログラムは簡潔になっている。しかしユーザが内部 DSL の状態を表現する `Reader` オブジェクトに触れる必要がある点は変わっていない。

### 5.4 SugarJ

SugarJ [3] はシンタックスシュガーをライブラリとして提供することのできる Java 拡張言語である。Lisp や Template Haskell [10] などが持つ構文マクロとは異なり、ホスト言語が構文解析できないような新しい文法をシンタックスシュガーとして追加できる。シンタックスシュガーを定義したライブラリをインポートすることで文法を変えることができる。ただし、SugarJ ではインポート文をファイルの先頭以外には記述できないため、ファイル内でその目的に応じて文脈を変化させるといったことはできない。

### 5.5 ユーザ定義演算子

Mixfix operators [2] は強力な表現力を持つユーザ定義可能な演算子の一種である。これは演算子の名前部分と引数部分が交互に並んでいるような演算子を指す。Mixfix operators や protean operators は様々な文法を表現できるため内部 DSL の実装に有用である。またこれらの多くは解析に型情報を利用するため、型が不正な内部 DSL をコンパイル時に検出することもできる。しかし、これらに対するモジュール機構は単純なものしか存在しておらず、文脈に応じて文法を自動的に切り替えるようなことはできなかった。本研究は protean operators に対するより良いモジュール機構を提案するものである。

### 5.6 Wyvern

Wyvern [9] は任意のユーザ定義リテラルを実現できるプログラミング言語である。Wyvern はユーザ定義リテラルの解釈に型情報を利用するので、型が不正な内部 DSL をコンパイル時に検出できる。ただし、Wyvern ではユーザ定義リテラルを使用する際に generic literal と呼ばれる特殊な構文を使わなければならないという制限がある。これは内部 DSL の構文の自由度を下げてしまい、簡潔性が低下する。

## 6 まとめ

内部 DSL の文脈情報を文脈オブジェクトと呼ぶオブジェクトとして表現し、それを暗黙的にやりとりする、内部 DSL の文脈を表す言語機構を提案した。本提案機構を用いることで、内部 DSL の状態をユーザの変数を介さずに、続く内部 DSL の式に渡すことができるため、簡潔で安全性の高い内部 DSL の実装に役立てることができる。文脈オブジェクトは内部 DSL の式により登録解除することもでき、その文脈オブジェクトを利用した内部 DSL は登録から登録解除の間でのみ利用できる。そのため、本提案機構は内部 DSL のスコープとその内部でのローカルな状態を表現するものであるとみなすこともできる。

我々はこのような言語機構をコンパイル型の静的型付き言語上に実装できるよう設計した。`activates` 節及び `deactivates` 節によって、追加・削除される文脈オブジェクトの型情報をコンパイラが静的に取得可能にし、また、どのパスを通った場合でも文脈に含まれる文脈オブジェクトの型が同じでなければならない、という制約を課すことで、静的な

解析を可能とした。現在、我々は前身研究である ProteaJ に提案した言語機構を追加したプログラミング言語 ProteaJ2 の実装を行っている。ProteaJ2 では提案した言語機構に加え、これをより使いやすくするために、`requires` 節、`propagate` 修飾子、`auto-close` 修飾子を実装することを予定している。最後に、提案した言語機構が有用に働くことを示すために、ProteaJ2 上で実装した内部 DSL の具体例を紹介した。

## 参考文献


[1] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, Sept. 2009.

[2] N. A. Danielsson and U. Norell. Parsing mixfix operators. In *Proceedings of the 20th international conference on Implementation and application of functional languages*, IFL'08, pages 80–99, 2011.

[3] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 391–406, 2011.

[4] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 111–122, 2004.

[5] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[6] K. Ichikawa and S. Chiba. Composable User-defined Operators That Can Express User-defined Literals. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 13–24, 2014.

[7] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[8] C. Omar, B. Chung, D. Kurilova, A. Potanin, and J. Aldrich. Type-directed, whitespace-delimited parsing for embedded dsls. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 8–11, 2013.

[9] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely Composable Type-Specific Languages. *ECOOP*, 2014.

[10] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, 2002.