日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

ユーザ定義演算子による内部DSLの構成法

市川 和央　千葉 滋

Internal Domain Specific Language (内部 DSL，内部ドメイン専用言語) [2] は強力な表現力を持つ DSL を汎用言語上で再現する試みであり，Scala [3] や Ruby をはじめとする多くの言語上に内部 DSL を実現するための機構が備えられている．しかし，内部 DSL を構築する従来のシステムにはいくつかの問題点があった．例えば，Lisp のようなマクロを用いた伝統的な手法は，プログラムの挙動のトレースや DSL のエラー報告を妨げてしまう．また，近年内部 DSL によく用いられる Scala や Ruby で用いられる手法は利用可能な文法に制限があり，実現できる DSL が限定されてしまう．そこで，我々は静的な型によりスコープが制限されたユーザ定義演算子により内部 DSL を構成する手法を提案する．この手法では式文の形の DSL を実現することができる．マクロのようなソースコードの変換を行わず，また DSL の文法にはほとんど制限がないので，従来のシステムのような問題が発生しない．

1　はじめに

Internal Domain Specific Language (内部 DSL, 内部ドメイン専用言語) はホスト言語 (汎用言語) の内部で構築された DSL を指し，これらは通常ライブラリやフレームワークとしてホスト言語上に実装される．内部 DSL はホスト言語の表現力を大きく向上させるため，Ruby や Scala のようなモダンな言語の多くは内部 DSL を実装するための有用なメカニズムを提供している．

しかし，Ruby や Scala などが提供する言語サポートによる手法は DSL に利用可能な文法に制限がある．Lisp のマクロや compile-time reflection [1] をはじめとするプログラム変換に基づく手法も非常に有用であるが，これらの手法も大きな問題を抱えている．DSL 開発者は抽象構文木などの言語の低レベルな部分に触れる必要があり，DSL 利用者は DSL のコンパイルエラーや実行時エラーを解釈することが困難となってしまう．

本論文では，静的な型によって制限されたユーザ定義演算子によって内部 DSL を構築する手法を紹介する．この手法では，DSL 開発者はソースコード変換を記述する必要がなく，また Ruby や Scala よりも柔軟な文法の DSL を記述することができる．ただし演算子によって DSL を表現するため，式以外の文法構造は表現できない．

2　静的な型で制限されたユーザ定義演算子

本論文ではプログラマーに新しい N 項演算子を定義することを許すことで，式の形の内部 DSL の開発を支援する手法について述べる．DSL のホスト言語は Java 言語とする．本手法は C++などの演算子オーバーロードと似ているが，我々の手法はすでに存在する演算子に限らず，どのような演算子でもユーザが自由に定義することができる．また，オペランドを String リテラルとして読むことを示す，readas 修飾

```
1 select name from register where age < 30
```

図 1　単純な SQL の SELECT 文の例

Constructing internal DSLs by user-defined operators
Kazuhiro Ichikawa　Shigeru Chiba, 東京工業大学大学院数理・計算科学専攻, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology.

```
1 (a) ResultSet rs = select name from register where age < 30;
2 (b) ShellScript script = java QuickSort < register;
```

図 2 DSL を含むプログラム例

子を提供する．演算子は返り値の型とオペランドの型でオーバーロードすることが可能で，その返り値の型が期待される型と一致した場合にのみ利用される．

DSL の文法の大部分はユーザ定義 N 項演算子の組み合わせにより表現することができる．例えば，図 1 のソースコードは SQL の SELECT 文の簡単な例である．この文は三項演算子 select...from...where... と二項演算子...<... の組み合わせであると解釈することができる．そのため，この文は三項演算子 select from where を定義し，二項演算子 < をオーバーロードすることにより表現することができる．このことから，本手法により図 2(a) のようなプログラムを記述できる．ここで，三項演算子 select from where の一つ目のオペランド name は列名であり変数名ではないが，readas 修飾子を用いることにより String リテラルとして読むことが可能となっている．

演算子はその返り値の型が期待される型と一致したときにのみ利用されるので，複数の DSL を同時に利用することができる．例えば図 2(b) のコードは単純なシェルスクリプト DSL を含む (少々わざとらしい) プログラム例である．代入式の右辺の式に期待される返り値の型は ShellScript であるため，二項演算子 < はリダイレクトであると判断できる．一方，SELECT 文の例では，age < 30 は where 句のオペランドの型であることが期待されるので，二項演算子 < はまったく異なるものとして解釈される．

本手法では，プログラマは抽象構文木のような言語の低レベルな部分を意識することなく，高いレベルのレイヤーで DSL を開発することができる．プログラム変換に基づかない手法なので，エラーメッセージの解釈が困難となってしまうこともない．また DSL に利用可能な文法もホスト言語にほとんど制限されないため，従来の手法よりも強力な DSL を実現することができる．

3 LasticJ

LasticJ は Java 言語のサブセットに前節で紹介したアイデアを導入した言語である．簡単のため，インターフェースや内部クラス，アノテーションやジェネリクスはサポートしていない．また制御構造も if, while，for，return のみをサポートしている．

N 項演算子の定義は operators 宣言の中に記述する．図 3 は operators 宣言の例である．このコードは三項演算子 select from where と二項演算子 < の二つの演算子を定義した，SQLOperators というモジュールの定義となっている．このようにして定義した operators モジュールは，クラスなどと同様に同一パッケージ内か，using 節 (クラスにおける import 節) により宣言されたもののみが有効となる．

N 項演算子の定義は Java 言語におけるメソッド定義と似ており，返り値の型，演算子のパターン，オペランドの型，演算子優先順位およびオペレータボディからなる．演算子のパターンは演算子のキーワードを表すクォートの付いた単語と，オペランドを表すクォートの付いていない識別子からなる．オペランドの型はメソッドのパラメータと同じように記述し，readas などもここに記述する．演算子優先順位は簡単のため今回の実装では正の整数値を指定するものとした．オペレータボディはメソッドボディと同様で，LasticJ 言語により行うべき処理を記述する．現在の実装ではすべてのユーザ定義演算子は左結合となっている．

LasticJ の構文解析アルゴリズム

LasticJ コンパイラの現在の実装では，構文解析器は LL バックトラックパーサとなっている．構文解析器は式を見つけるとその部分で有効な演算子を探し，見つかった演算子を利用するものと仮定して構文解析を行う．もし複数の演算子が有効であったならば，

```
1  operators SQLOperators {
2          ResultSet 'select col 'from table 'where cond
3           (readas SQLColumn col, SQLTable table, SQLCond cond)
4           : priority = 100
5          {
6                  Connection conn = ...;
7                  PreparedStatement stmt = conn.prepareStatement(...);
8                  return stmt.executeQuery();
9          }
10
11         SQLCond col '< val (readas SQLColumn col, int val)
12          : priority = 200
13         {
14                 ...
15         }
16 }
```

図 3 N 項演算子の定義例

それらすべてを試す．すべての演算子について構文解析が失敗した場合，通常の Java 言語の文法による構文解析を行う．

演算子はその返り値の型が期待される型と一致した場合にのみ利用されるので，構文解析器は有効な演算子として式の期待する型 (またはそのサブタイプ) を返すような演算子を選択する．例えば，図 2(a) の例ならば，右辺の式は ResultSet 型を返すことが期待されるので，ResultSet 型を返す三項演算子 select from where が選択され，age < 30 の部分は SQLCond 型を返すことが期待されるので，図 3 で定義しているような二項演算子 < が選択される．

構文解析器は選択された各々の演算子に対して，その演算子のパターンが構文ルールに含まれているかのようにして解析を行う．また，同時に字句ルールもその演算子のキーワードを含むよう変更される．演算子のオペランドに readas 修飾子が付けられていた場合，そのオペランドは一旦 String リテラルとして読まれ，その後コンストラクタに代入されることで適当な型のリテラルのように解釈される．

通常，オペランドの位置には式が入るため，このアルゴリズムは再帰的に適用される．その際，有効な演算子は演算子優先順位を考慮して選択される．

選択されたすべての演算子について構文解析が終了したとき，ただ一つの演算子のみが構文解析に成功していたならば，コンパイラはその解析結果を採用する．もしすべての演算子について構文解析が失敗していたならば，コンパイラはその式が通常の Java の式であるものとして再度構文解析を行う．構文解析に成功した演算子が複数存在した場合，コンパイラはその部分の式が曖昧であることを示すエラーを投げる．コンパイラが潜在的に曖昧さを含むような演算子の定義を許容する点には注意が必要である．曖昧性に関するエラーは実際に構文解析をするときにしか検出できない．

以上のように，構文解析器は期待される型をもとに解析規則を決定するので，抽象構文木は根から順に再帰的に作り上げていく必要がある．ゆえに，構文解析アルゴリズムはトップダウン方式でなければならないため，今回は LL バックトラックパーザを利用した．

最終的に，図 3 の演算子定義のもとで，図 2(a) の例は図 4 のように解釈される．ここで，`select$from$where$` は三項演算子 select from where に，`$lessthan$` は二項演算子 < に対応する演算の呼び出しを関数として表記したものである．

4 関連研究

N 項演算子を利用して内部 DSL を構成するというアイデアは新しいものではなく，今までにも多くの言語で試みられてきた．C++言語は演算子オーバー

```
1 ResultSet rs = select$from$where$(new SQLColumn(``name''), register,
2                                   $lessthan$(new SQLColumn(``age''), 30));
```

図 4 擬似コードによる図 2(a) の解釈

ロードの機構を持っており，既存の演算子の大部分はオーバーロード可能となっている．C++言語の標準ライブラリはこの機構を利用することでよりわかりやすく簡潔な記述を可能としている．しかし，C++言語の演算子オーバーロードではユーザが新しい演算子を定義することはできない．Haskell も C++言語と同様演算子オーバーロードの仕組みを持っているが，こちらはユーザが新しい演算子を定義することを許す．ただし，定義可能な演算子は二項演算子に限られ，また演算子名は記号のみで構成されていなければならない．Scala ではメソッド呼び出しを単項演算または二項演算のように記述することができる．単項演算と二項演算を組み合わせることで N 項演算をエミュレートすることが可能だが，演算子優先順位が存在しないため実現可能な構文に制限がある．Smalltalk のメッセージ式は N 項演算のような形で記述される．しかし Smalltalk は動的型付け言語であるためメッセージ式の解析の際に型は参照されず，また Scala と同様に実現可能な構文に制限がある．

プログラム変換は内部 DSL の構成方法としてよく知られており，Lisp のマクロや compile-time reflection をはじめとして非常に多くの手法が知られている．Template Haskell [5]，Nemerle [6]，Converge [7]，Helvetia [4] はプログラム変換により内部 DSL を構成することができる言語の一例である．これらの言語は非常に強力な DSL を実現できるが，そのためには抽象構文木のような言語の低レベルな部分を操作する必要がある．

5 まとめと今後の課題

本論文ではユーザ定義 N 項演算子を利用した内部 DSL の構成方法を提案し，それを実現する機能を持つ言語として LasticJ を紹介した．演算子定義により文法を拡張するため，抽象構文木のような低レベルな部分に触れることなく内部 DSL を実現することができる．また，DSL に利用可能な文法もホスト言語にほとんど制限されないため，従来の手法よりも強力な DSL を実現することができる．

現在の LasticJ の構文解析器はバックトラックパーサとなっているため，コンパイルに大きなオーバーヘッドがかかる．今後の課題として，このオーバーヘッドが具体的にどの程度であるかをさまざまな DSL を実装して計測し，このシステムが実用に耐えうるかどうかを判断することが挙げられる．

現在の実装ではオプションや 0 回以上の繰り返しの表現ができないため，余分な演算子やクラスの定義が必要になってしまうことがある．そのため，正規表現のような形でこれらを表現できるようにすることも今後の課題の一つである．

参 考 文 献

[1] Chiba, S.: A metaobject protocol for C++, *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, ACM, 1995, pp. 285–299.

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

[3] Odersky, M., Spoon, L., and Venners, B.: *Programming in Scala*, Artima Press.

[4] Renggli, L., Gîrba, T., and Nierstrasz, O.: Embedding languages without breaking tools, *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, Springer-Verlag, 2010, pp. 380–404.

[5] Sheard, T. and Jones, S. P.: Template meta-programming for Haskell, *SIGPLAN Not.*, Vol. 37(2002), pp. 60–75.

[6] Skalski, K., Moskal, M., and Olszta, P.: Meta-programming in Nemerle, `http://nemerle.org/metaprogramming.pdf`.

[7] Tratt, L.: Domain specific language implementation via compile-time meta-programming, *ACM Trans. Program. Lang. Syst.*, Vol. 30(2008), pp. 31:1–31:40.