動的型付け言語における構文拡張による曖昧性の回避手段の提供

岩間 雄太* 市川 和央† 千葉 滋‡
東京大学大学院情報理工学系研究科創造情報学専攻*†‡

1 はじめに

内部 DSL を実現する機構には文法に制約がある．文法の制限を緩めると曖昧性といわれる構文解析時に一意に解析結果が決まらなくなる問題が発生するためである．静的型付き言語では型の情報を使用することで曖昧性の問題を回避することが可能だが動的型付き言語では同じ解決策を取ることができない．本稿では同一 DSL の演算子のオペランド式の中でだけ使用できる演算子を動的型付き言語に導入することを提案する．本提案によりグローバルに見えている演算子の数を減らすことが可能になるため，強力なユーザ定義演算子をサポートしつつ，曖昧性の問題が起こりにくくなる．

2 構文拡張時に発生する曖昧性の問題

動的型付き言語上で，多くの構文拡張をするためには構文解析上の曖昧性を解決する必要がある．構文解析上の曖昧性とは，プログラムを構文解析する際に複数の構文木が存在し，解析結果が一意に決まらなくなる問題である．以下では，構文拡張時に発生する曖昧性，および曖昧性に対する既存の解決法について述べる．

内部 DSL [1] はホスト言語の構文に依存しているため，内部 DSL には構文的な制約がある．内部 DSL とは，ホスト言語内に実装された特定の種類の問題を解くためのプログラミング言語である．ホスト言語の字句解析器，または構文解析器を拡張できる機構を導入すると，ホスト言語の構文に依存しない内部 DSL を実装することができる．しかし，安易にホスト言語のプログラマが構文を拡張できるようにすると，拡張した構文と元からホスト言語に備わっている構文，あるいは拡張した構文同士の間に構文解析上の曖昧性が発生する．構文拡張をするほど構文同士が衝突しやすくなり曖昧性が生じやすくなる．構文解析上の曖昧性が発生する具体的な例を示す．たとえばユーザが二つの if 演算子を定義したとする．一つ目は `if Expr Expr` のように if の後ろに `Expr` という非終端記号が来る演算子である．二つ目は `Expr if Expr` のように if の間に `Expr` が来る演算子である．それぞれの使用例を Listing 1 に示す．これらの二つの if が定義してある場合，Listing 2 のようなプログラムがあると，構文解析器はどちらの if を使用すればよいか一意に決定することができず，構文解析上の曖昧性が生じる．実際に，ユーザが構文拡張をすることのできるプログラミング言語である SugarJ [2] では，このような場合はコンパイルエラーとなる．

Listing 1: 二つの if の使用例

```
x = -10
if x.neg? (-1 * x) #=> 10

[1, 2 -1] if pos? #=> [1, 2]
```

Listing 2: 構文的な曖昧性が発生する例

```
loop [1, 2, -1] if pos? do |x|
  print x
end
```

Listing 3: 曖昧性回避のための型の定義

```
1. (if e: Bool, e: a): a
2. (e: List[a], if e: Bool): List[a]
```

静的型付き言語では型を利用することで構文解析上の曖昧性を解決することが可能なことが知られている．オーバーロードは引数の型が異なる同一の構文規則を持つ構文同士の曖昧性を解決する．これを利用することで，演算子オーバーロードと呼ばれる，ユーザによる演算子の追加を可能とする機構を提供することができる．ProteaJ [3] はこれを拡張し，返り値の型と引数の型の両方を使って構文同士の曖昧性を解決するようにした言語である．こうすることで，通常の演算子オーバーロードよりも強力な構文拡張を提供することができる．ProteaJ の場合 Listing 2 の曖昧性の問題は Listing 3 のように型を使用することで解決可能である．この例の場合，`[1, 2, -1] if pos?`の返り値の型が List[a] だと解釈でき，Listing 3 の二つの定義の戻り値の型はそれぞれ `List[a]` と `a` であるとわかっているので，1 の構文で構文解析すればよいことがわかる．しかしながら，動的型付き言語では型を使用した手法が使えないためこのような従来手法は利用できない．

Providing a method for avoiding ambiguities caused by syntactic extensions in a dynamically typed language
* Yuta Iwama †Kazuhiro Ichikawa ‡Shigeru Chiba
*†‡ Graduate School of Informatics Science and Technology the University of Tokyo

3 動的型付き言語上での曖昧性の回避方法の提案

本稿は指定の文脈名をもった演算子のオペランドのみで使用できる演算子をプログラム中で定義できるようにすることで構文の曖昧性を回避しつつ，プログラマが構文拡張できるようにする手法を提案する．2 章で述べたように，動的型付き言語では静的型付き言語のようにコンパイル時に型を使用して構文解析上の曖昧性を解決することはできない．そこで，二種類の演算子を定義できるようにして構文解析上の曖昧性を回避する．一つ目の種類の演算子はグローバルなスコープで使用することのできる演算子である．二つ目の種類の演算子は指定の文脈名を持った一つ目の演算子のオペランドでのみ使用できる演算子である．二つ目の演算子は，特定の文脈名をもった演算子のオペランドのみで使用できるので，グローバルなスコープでは使用できない．DSL 開発者は，二つ目の演算子をなるべく使用してグローバルな構文拡張の数を減らすことが可能である．本提案によりグローバルなスコープから見える演算子の数が減り，曖昧性の問題を回避しながら，強力な構文拡張が可能となる．

たとえば，Listing 2 は本提案のオペランドの中のみで使用できる演算子を使用すれば Listing 4 のように曖昧性を回避できる．まず `loop _ _`というグローバルに使用することができる演算子を定義する．その後，定義した loop のオペランド内でのみ使用できる if を定義する．このように定義をすると型を使用しなくても，`loop` のオペランドでのみ使用できる if を構文解析時に使用できる．以上のようにして 2 章で示した曖昧性を回避することができる．

4 予備的な実装実験

Listing 3 を実現するために `defi` と `defop` という言語機構を，我々が作成した OCaml ライクな構文を持つ言語に対して実装した．`defi` は新しい演算子をグローバルに定義することができる言語機構である．`defop` は `defi` で定義された演算子のオペランドのみで使用可能な演算子を定義することができる言語機構である．Listing 5 は loop および if の定義例である．それぞれ `defop` および `defi` を使用して定義した．ここで，ENUM は文脈名を表す．よって，この if は `ENUM` という文脈名を持った演算子のオペランドのみで使用することができる．また，コンマ (,) が付いている引数はオペランドを表している．

Listing 4: 本提案の手法を使って曖昧性を回避した例

```
loop [1, 2, -1] if pos? do |x|
  print x
end
```

Listing 5: 実装した実装の使用例

```
defi[ENUM](loop ,expr ,expr) { ... }
defop[ENUM] (,expr if ,expr) { ... }
```

本言語のインタープリタは Scala で実装した．DSL 開発者により動的に構文規則を拡張できる必要があるので，パーザコンビネータを使用し構文解析器を組み立てるようにした．また，Packrat Parsing [4] を採用することで，再帰下降構文解析では最悪で指数時間かかる解析を線形時間で行うことを可能にした．

5 終わりに

本稿では，動的型付き言語で，多くの構文を利用可能にするためには曖昧性を解決する必要があることを示した．また，発生する曖昧性に対して，指定した文脈名を持った演算子のオペランドのみで使用できる演算子を定義できるようにすることで解決する手法の提案をした．

参考文献

[1] Martin Fowler. InternalDslStyle. `http://martinfowler.com/bliki/InternalDslStyle.html`, 2006.

[2] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proc. of OOPSLA 2011*, pp. 391–406. ACM, 2011.

[3] Kazuhiro Ichikawa and Shigeru Chiba. Composable user-defined operators that can express user-defined literals. In *Proc. of MODULARITY '14*, pp. 13–24. ACM, 2014.

[4] Bryan Ford. Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pp. 36–47. ACM, 2002.