環境にメソッドを直接格納する新しいオブジェクトシステムの提案

林 拓人[1,a)]　前田 敦司[2,b)]

概要：従来のクラスベース・オブジェクトシステムは，メソッドをクラスに格納するものと総称関数に格納するもの（メソッドがクラスに属すものと総称関数に属すもの）とに分けられる．本論文はこれらのいずれとも異なり，環境にメソッドを直接格納する新しいオブジェクトシステムを提案する．クラスや総称関数といった枠を廃し，クラス名とメソッド名の組をキーとして環境にメソッドを直接格納する．これにより変数に対して行えるあらゆる操作がメソッドに対しても自然に行えるようになり，従来の方式に比べシンプルな仕組みで柔軟なオブジェクト指向プログラミングが可能となる．提案する手法の有用性を実証するため，このオブジェクトシステムを搭載する独自言語 Suzu の処理系を実装した．Suzu の特徴を生かすプログラム例として言語内 DSL（Domain Specific Language）の構築例を挙げる．従来のオブジェクトシステムにおける類似した概念等との関連についても議論する．

キーワード：オブジェクトシステム，メソッド，クラス，総称関数，環境

1. 序論

従来のクラスベース・オブジェクトシステムは，メソッドの格納方式をモデル化すると次の 2 つのモデルに分類することができる．1 つはクラスにメソッドを格納するモデル，もう 1 つは CLOS[6] のように総称関数にメソッドを格納するモデルである．

本論文は，これら 2 つの格納モデルに対する分析に基づき考案した，全く新しい格納モデルを持つオブジェクトシステムを提案する．また，その特徴を最大限に生かせるよう独自に設計したプログラミング言語 Suzu を用いて，提案するオブジェクトシステムの評価を行う．

なお，ここでのモデルとはプログラミング言語が仕様上オブジェクトシステムをどうとらえているかを表すものであり，特定の実装方法を指すものではない．オブジェクトシステムに対し従来とは異なる新たな見方を提供することで，有用と思われる機能に自然な意味付けを与えたり，既存の概念を整理したりといった目的を持つものである．

2. 従来のオブジェクトシステム

ここではモデルを単純化するため，単一ディスパッチ（1 つのオブジェクトに基づきメソッドを決定する方式）の場合についてのみ考える．多重ディスパッチ（複数のオブジェクトに基づきメソッドを決定する方式）へのモデルの拡張については 6.2 節で検討する．

[1] 筑波大学情報学群情報科学類
[2] 筑波大学システム情報系
[a)] hayashi@ialab.cs.tsukuba.ac.jp
[b)] maeda@cs.tsukuba.ac.jp

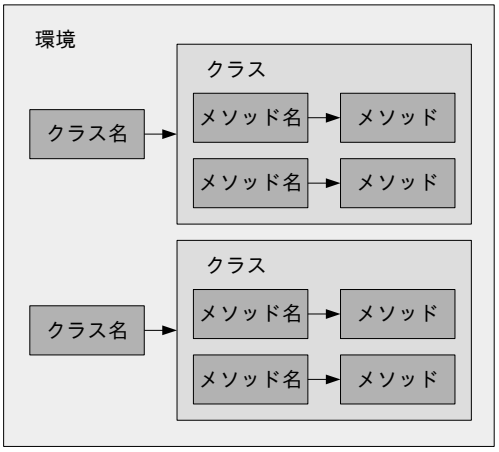

図 1 クラスにメソッドを格納するモデル

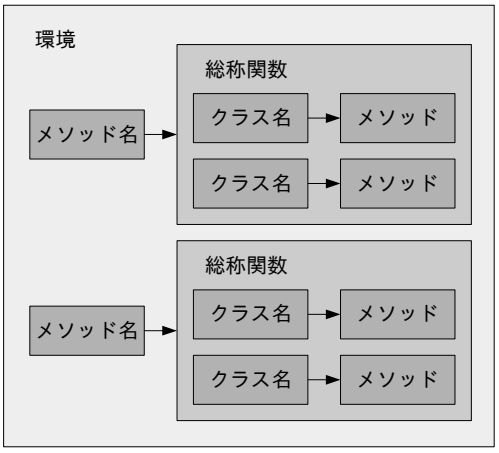

図 2 総称関数にメソッドを格納するモデル

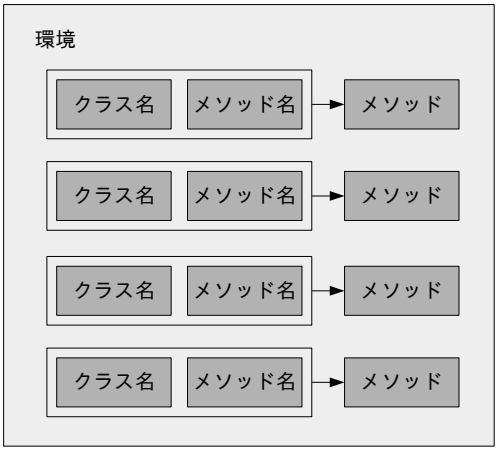

図 3 環境にメソッドを直接格納するモデル

2.1 クラスにメソッドを格納するモデル

クラスにメソッドを格納するモデルでは環境（名前と値の束縛の集合）にクラスを格納し，クラスにメソッドを格納する（図 1）．ここで環境はクラス名をキーとしてクラスを格納する辞書であり，クラスはメソッド名をキーとしてメソッドを格納する辞書である．Smalltalk や Ruby といった言語のオブジェクトシステムはこのモデルに分類される．

2.2 総称関数にメソッドを格納するモデル

総称関数にメソッドを格納するモデルでは環境に総称関数を格納し，総称関数にメソッドを格納する（図 2）．ここで環境はメソッド名をキーとして総称関数を格納する辞書であり，総称関数はクラス名をキーとしてメソッドを格納する辞書である．単一ディスパッチに限定した CLOS はこのモデルに分類される．

3. 提案するオブジェクトシステム

従来のオブジェクトシステムの分類先である 2 つのモデルでは，どちらもクラス名とメソッド名が決まればメソッドが一意に定まることが要請されている．また，環境という辞書の中にクラスあるいは総称関数という辞書が入れ子になっている構造も共通している．

提案するオブジェクトシステムはこのようなクラスや総称関数による入れ子構造を廃し，**環境にメソッドを直接格納するモデル**を採用する（図 3）．ここで環境は**クラス名とメソッド名の組**をキーとしてメソッドを格納する辞書である．

このモデルの特徴は，メソッドの格納方式が一般的な変数の格納方式と類似していることである．メソッドがクラス名とメソッド名の組をキーとして環境に格納されるのに対し，変数は変数名をキーとして環境に格納される．

これはすなわち，「変数名」を「クラス名とメソッド名の組」に置き換えることで，**変数に対して行えるあらゆる操作がメソッドに対しても自然に行える（自然な意味付けが可能である）**ということである．具体的には，ローカル変数に対応する**ローカルメソッド**の定義，シャドーイング，モ

ジュールからのエクスポート・インポート，仮引数としての指定などが挙げられる．4 節では提案するオブジェクトシステムを搭載した独自のプログラミング言語 Suzu を用いて，この特徴を生かした実際のプログラム例を示す．

4. 実装：プログラミング言語 Suzu

環境にメソッドを直接格納するモデルに基づき設計した独自のプログラミング言語 Suzu の解説とそのプログラム例により，提案するオブジェクトシステムの有用性を示す．

4.1 基本的な文法

//から行末まではコメントである．以降，//=> に続くコメントはプログラムの出力を表すものとする．出力にはデバッグ用出力関数 p を用いる．

変数を定義するには let x = 123 のようにする．ここでは変数 x に整数値 123 を代入している．

関数呼び出しは func(arg1, arg2) のようにする．ここでは関数 func を第 1 引数 arg1，第 2 引数 arg2 で呼び出している．

関数定義は以下のようにする．

```
def fac(n):
  if(n == 0):
    1
  else:
    n * fac(n - 1)
  end
end
```

これは以下のコードと意味的に等価である．

```
let fac = ^(n):
  if(n == 0):
    1
  else:
    n * fac(n - 1)
  end
end
```

^(n):から end までの部分は引数 n を受け取って end までの式を評価する関数リテラルである．^(n){ ... }と書くこともできる．

関数リテラルは関数呼び出しの後ろに続けて書くことで追加の引数として高階関数に渡すことができる．例えば

```
map(lst)^(n):
  n * 2
end
```

は

```
map(lst, ^(n){ n * 2 })
```

と等価である．

begin は新たなスコープを導入し end までの式を評価する．

```
let str = "global"
begin:
  let str = "local"
  p(str)  //=> "local"
end
p(str)  //=> "global"
```

Suzu では:と end（または{と}）で囲まれた箇所を**ブロック**と呼ぶ．ブロックには新たなスコープが導入される．

4.2 オブジェクトシステム

ここでは 3 節で提示したモデルの Suzu における具体的な実装を述べる．

クラス名とメソッド名の組は C#m と書く．C はクラス名，m はメソッド名である．

クラス定義は以下のようにして行う．

```
class Vector:
  def MkVector(x, y)
end
```

これにより，クラス Vector とそのコンストラクタ MkVector が定義される．

メソッドは以下のようにクラスとは独立して定義する．仮引数ではコンストラクタ等を用いたパターンマッチングが使用できる，第 1 引数はメソッド呼び出しの対象となったオブジェクト自身，第 2 引数以降は与えられた実引数となる．

```
def Vector#add(MkVector(x1, y1),
               MkVector(x2, y2)):
  MkVector(x1 + x2, y1 + y2)
end
```

これは通常の関数定義における変数名をクラス名

とメソッド名の組に置き換えた形となっている．変数と同様 let を用いた書き方も可能である．

メソッドは obj.m(arg1, arg2) のようにして呼び出す．先程述べたように，メソッドとして呼び出される関数には第 1 引数として obj，第 2・第 3 引数として arg1，arg2 が渡される．以下の a.add(b) という呼び出しでは上の例で定義した Vector#add に，第 1 引数として a，第 2 引数として b が渡される．

```
let a = MkVector(1, 2)
let b = MkVector(3, 4)
p(a.add(b))  //=> MkVector(4, 6)
```

また，独自の演算子をメソッドとして定義することもできる．二項演算子の場合，左辺がメソッド呼び出しの対象となるオブジェクト，右辺が実引数となる．

```
let Vector#(+) = Vector#add
p(a + b)  //=> MkVector(4, 6)
```

Suzu は 3 節で述べた環境にメソッドを直接格納するモデルを採用しているため，ネストしたスコープの内側にメソッドを定義することで，ローカル変数に対応するローカルメソッドをごく自然に定義できる．

```
let v = MkVector(5, 6)
def Vector#m(self):
  p("global")
end
begin:
  def Vector#m(self):
    p("local")
  end
  v.m  //=> "local"
end
v.m  //=> "global"
```

このように，メソッドの探索は内側のスコープから順に行われるため，メソッドを変数のようにシャドーイングすることが可能である．

メソッド呼び出しの手順を図に表わすと図 4 のようになる．a.f(...) というメソッド呼び出しは次の 3 ステップを経て実行される．

(1) a からクラス名 A を取り出してメソッド名 f と組にする

(2) 組をキーとして環境をルックアップしメソッド<method>を発見

(3) 第一引数に a を追加し<method>を呼び出す

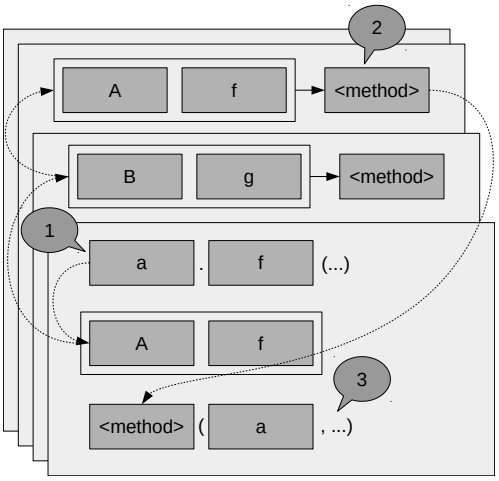

図 4 メソッド呼び出しの手順

4.3 モジュールとトレイト

Suzu のオブジェクトシステムと調和するモジュールシステムについて述べる．

Suzu のモジュールは他の言語における変数のようにメソッドを個別にエクスポートできる．

```
module A:
  def f(...):
    ...
  end
  def C#m(...):
    ...
  end
  def C#n(...):
    ...
  end
  export f, C#m
end
```

上の例ではモジュール A は変数 f とメソッド C#m をエクスポートするが，C#n はエクスポートしない．エクスポートされた変数やメソッドは::を用いて，A::f，A::(C#m) のように参照できる．

モジュールは open キーワードを用いて open す

ることができる．モジュールを open するとエクスポートされている変数やメソッドがそのスコープで直接定義されたようにインポートされる．また，except キーワードを用いてインポートの対象から除外することもできる．

```
module B:
  ...
  export f, g, C#m, C#n
end
begin:
  open B except g, C#n
  ...
end
```

上の例では begin から end までのスコープで，モジュール B でエクスポートされているもののうち g と C#n を除いてインポートを行っている．これによりスコープ内で f と C#m を B::という修飾子なしに参照できるようになる．

Suzu は継承機構を持たず，実装の再利用はトレイト [9] を用いて行う．ただし，Suzu のトレイトはそのオブジェクトシステムに適合するよう他の言語と比べて多少趣の異なるものとなっている．

```
trait T(C, C#m, C#n):
  def C#o(self, ...):
    ...
    self.m(...)
    ...
  end
  def C#p(self, ...):
    ...
    self.n(...)
    ...
  end
  export C#o, C#p
end
```

トレイトはクラスやメソッドを受け取ってモジュールを返す関数として表現される．上の例では C#m や C#n のように仮引数としてメソッドを指定している．使用する側は以下のようにする．

```
open T(D, D#m, D#n)
open T(E, E#m, E#n)
```

トレイトは既存のクラスとメソッドを受け取って新たなメソッドが定義されたモジュールを生成する．これを open することでメソッドがそのスコープで有効になる．この例では，D#o，D#p，E#o，E#p が新たに定義される．

トレイト同士の加算は単に open を並べればよい．メソッドの減算は except を用いる．メソッドのリネームは except に let を用いた個別のインポートを組み合わせる．例えばトレイト S によって定義されるメソッド m を n にリネームしたい場合，以下のようにする．

```
open S(C) except C#m
let C#n = S(C)::(C#m)
```

4.4 プログラム例：PEG パーザコンビネータ

Suzu の有用性を示すプログラム例として，ここまで述べてきたオブジェクトシステムとモジュールシステムを活用する PEG パーザコンビネータライブラリを作成した．ソースコードは付録の A.1 節に記載している．

PEG（Parsing Expression Grammar）とは，形式言語を表す曖昧さのない文法定義である．このライブラリを用いると，PEG パーザを言語内 DSL として簡潔に記述することができる．例えば，文脈自由でない言語 $\{a^n b^n c^n \mid n \geq 1\}$ を表す PEG

```
S <- &(A !'b') 'a'+ B !'c'
A <- 'a' A? 'b'
B <- 'b' B? 'c'
```

に基づくパーザは，トレイト PEG::New を使用して以下のように書ける．

```
let s = begin:
  open PEG::New()
  "S" <- and(nt_ref("A") &+
             not(char('b')))
      &+ one_or_more(char('a'))
      &+ nt_ref("B")
      &+ not(char('c'))
  "A" <- char('a')
      &+ zero_or_one(nt_ref("A"))
      &+ char('b')
  "B" <- char('b')
```

```
        &+ zero_or_one(nt_ref("B"))
        &+ char('c')
    nt_ref("S")
  end
```

PEG::New の呼び出しの戻り値を open することで，文法定義のためのメソッド・関数群が現在のスコープにインポートされる．例えば，(<-) は非終端記号の定義，nt_ref は非終端記号の参照，(&+) は連接，(|+) は選択（例では使用していない），and は肯定先読み，not は否定先読みである．パーザは関数 PEG::parse を使用して実行できる．

```
p(PEG::parse(s, ""))
  //=> Failure()
p(PEG::parse(s, "abc"))
  //=> Success(...)
p(PEG::parse(s, "ab"))
  //=> Failure()
p(PEG::parse(s, "aaabbbccc"))
  //=> Success(...)
p(PEG::parse(s, "aabbbccc"))
  //=> Failure()
```

このライブラリは，**既存のデータ型に対して新たな演算子をローカルに定義できる**という Suzu の特徴を生かしている．演算子 (<-) の正体はメソッド String#(<-) であり，スコープを限定した上で既存の文字列クラス String に対し新たに定義されている．

このように，Suzu のオブジェクトシステムは特定のスコープでのみ有効なメソッドを定義できることで，グローバル環境を汚染しない可読性の高い言語内 DSL の作成を可能にしている．

5. 関連研究

従来のオブジェクトシステムに Suzu と似た柔軟性を与える取り組みや，構造上の類似性を持つ概念との比較を行う．

ContextJ[1] は，文脈指向プログラミングにおける layer という概念に基づいた Java の拡張言語である．layer を切り替えることによりメソッドの定義を切り替えられる点が Suzu のモジュールと類似しているが，ContextJ のメソッド定義は Java と同様クラスの内部でしか行えない．

GluonJ[4] は，Java でアスペクト指向プログラミングを行うためのシステムである．Glue と呼ばれるクラスを定義することで既存のクラスの外部でメソッドを再定義できるが，再定義の影響範囲は後述の Classboxes[3] と違いグローバルである．

Classboxes は，影響範囲を Classboxes というモジュール内に制限してメソッドを再定義できるシステムである．Suzu は Classboxes のようなクラス専用の特殊なモジュールを使うことなく，変数を扱うのと同じモジュールシステムによって再定義の影響範囲を制限できる．また，Classboxes はダイナミックスコープだが，Suzu のモジュールはレキシカルスコープである．

Method Shells[10] は，link と include という 2 種類の宣言を使い分けることによって，ダイナミックスコープとレキシカルスコープを使い分けるモジュールシステムである．先ほど述べたように Suzu はレキシカルスコープのみサポートする．これは一般的な変数の扱いと同様にすることで理解を容易にするためである．

Refinements[7] は，プログラミング言語 Ruby に導入された Classboxes と類似する機構である．Refinements はレキシカルスコープである点で Suzu のモジュールシステムにより近い．しかしながら，Refinements はメソッドの再定義の有効・無効をファイル単位でしか制御できない．Suzu はより細かいブロック単位での制御が可能である．

Suzu のオブジェクトシステムはデータ型の定義の外部でその値に対する演算子の振る舞いを変えられるという点で，型クラス [11] に類似している．型クラスは導入に静的な型を必要とするが，Suzu はこれを必要としない．また型クラスのインスタンス宣言はモジュールからのエクスポート・インポートによって可視性を制御できないが，Suzu のメソッド定義はこれが可能である．ただし型クラスは戻り値の型に応じて関数の振る舞いを変えさせることができるのに対し，Suzu ではこれは不可能である．

MixJuice[5] は，複数のモジュールにクラス定義を分割し組み合わせることで，プログラムのモ

ジュール化を促進するプログラミング言語である．MixJuice のモジュールシステムが与える柔軟性は Suzu のそれと類似している．違いは MixJuice が 2.1 節で述べたクラスにメソッドを格納するモデルを採用しているのに対し，Suzu は 3 節で述べた環境にメソッドを直接格納するモデルを採用していることである．

6. 今後の課題

6.1 継承機構

Suzu は継承機構を持たず，トレイトによって実装の再利用を行う．継承機構を持たないことによって生じる不都合については今後検討していく必要がある．

もし Suzu に継承機構を追加するならば，オブジェクトに複数のクラス名を持たせればよい．クラス名は先頭からメソッド解決の順に並べたリストで持つようにする．メソッド呼び出しの際はオブジェクトが持つリストの先頭から順にクラス名を取り出し，メソッド名との組を作ってこれをキーとし環境からメソッドの探索を行う．

単一継承に制限する場合，クラス名のリストは先頭が継承ツリーの最下位クラス，末尾が最上位クラスとなる．多重継承を許す場合，C3 線形化 [2] 等を用いて適切な順序で並べたクラス名のリストを生成する必要がある．

6.2 多重ディスパッチ

2.2 節で示した総称関数にメソッドを格納するモデルは，CLOS がサポートする多重ディスパッチに対応していない．モデルを多重ディスパッチに対応させるには，総称関数を単なる辞書ではなくある種のデータベースとしてとらえる必要がある．データベースはすべての引数のクラス名を受け取って，その組み合わせにマッチするメソッドを検索し返す．

Suzu は多重ディスパッチに対応していないが，この考え方を応用し対応させることが可能である．すなわち環境をある種のデータベースとしてとらえ，複数のクラス名と 1 つのメソッド名を受け取ってその組み合わせにマッチするメソッドを検索し返す．

つまり，1 つのクラス名と 1 つのメソッド名からメソッドが決まるのが単一ディスパッチ，複数のクラス名と 1 つのメソッド名からメソッドが決まるのが多重ディスパッチであると言える．ここで自然と，1 つのクラス名と複数のメソッド名または複数のクラス名と複数のメソッド名からメソッドが決まるシステムというのも思い浮かぶ．これらは筆者らの知る限り既存のオブジェクトシステムにない概念であり，考察の余地がある．

6.3 メソッド呼び出しの最適化

Suzu のオブジェクトシステムにはメソッド呼び出しの一般的な最適化手法 [8] がそのままでは適用できないことがある．これは Suzu がメソッドをローカルに定義できることや，関数の仮引数としてメソッドを指定できることによる．同じクラス名とメソッド名の組に対しても呼び出し位置が変われば呼び出されるメソッドが変わるほか，同じ位置においても関数呼び出しのたびにメソッドの内容が変わることもある．

Suzu には継承が無いためクラス階層のルックアップを省くための最適化は必要ない．代わりにローカルメソッドの効率的な呼び出し方法について検討する必要がある．現在 Suzu の処理系は特に最適化を施していないため，適切な最適化手法を考え実装することが課題である．

6.4 ダイナミックスコープの導入

Suzu のメソッドは現状レキシカルスコープのみサポートしている．ダイナミックスコープを導入する場合の導入方法とその応用例については検討の余地がある．Classboxes や Method Shells 等ダイナミックスコープをサポートするモジュールシステムとの，具体的なプログラム例を用いたより詳細な比較が，ダイナミックスコープ導入の是非を語る上で重要となってくるだろう，

7. 結論

従来のクラスベース・オブジェクトシステムはクラスにメソッドを格納するモデルと総称関数に

メソッドを格納するモデルとに分類されたが，そのどちらにも属さない，環境にメソッドを直接格納するモデルを持つ新しいオブジェクトシステムを提案した．

このモデルにおいては変数に対して行えるあらゆる操作がメソッドに対しても自然に行える．この特徴を生かし，提案するオブジェクトシステムを搭載した独自言語 Suzu を用いて可読性の高い言語内 DSL を作成し，提案手法の有用性を示した．

オブジェクト指向プログラミングにおける既存の様々な概念の整理にも役立った．今後はより実用性を意識した拡張や効率的な実装について考えていくことが課題である．

謝辞 有益なコメントを下さったコメンターの方々に感謝いたします．

参考文献

[1] Appeltauer, M., Hirschfeld, R., Haupt, M. and Masuhara, H.: ContextJ: Context-oriented Programming with Java, *Information and Media Technologies*, Vol. 6, No. 2, pp. 399–419 (online), DOI: 10.11185/imt.6.399 (2011).

[2] Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K. and Withington, P. T.: A Monotonic Superclass Linearization for Dylan, *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, New York, NY, USA, ACM, pp. 69–82 (online), DOI: 10.1145/236337.236343 (1996).

[3] Bergel, A., Ducasse, S., Nierstrasz, O. and Wuyts, R.: Classboxes: Controlling Visibility of Class Extensions, *Comput. Lang. Syst. Struct.*, Vol. 31, No. 3-4, pp. 107–126 (online), DOI: 10.1016/j.cl.2004.11.002 (2005).

[4] Chiba, S., Igarashi, A. and Zakirov, S.: Mostly Modular Compilation of Crosscutting Concerns by Contextual Predicate Dispatch, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, New York, NY, USA, ACM, pp. 539–554 (online), DOI: 10.1145/1869459.1869503 (2010).

[5] Ichisugi, Y. and Tanaka, A.: Difference-Based Modules: A Class-Independent Module Mechanism, *ECOOP 2002 - Object-Oriented Programming* (Magnusson, B., ed.), Lecture Notes in Computer Science, Vol. 2374, Springer Berlin Heidelberg, pp. 62–88 (online), DOI: 10.1007/3-540-47993-7_3 (2002).

[6] 井田昌之，元吉文男，大久保清貴（編）：Common Lisp オブジェクトシステム—CLOS とその周辺—，共立出版 (2010).

[7] 前田修吾：Refinements とは何だったのか，http://magazine.rubyist.net/?0041-200Special-refinement (2013).

[8] 小野寺民也：オブジェクト指向言語におけるメッセージ送信の高速化技法，情報処理，Vol. 38, No. 4, pp. 301–310（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110002769654/⟩ (1997).

[9] Scharli, N., Ducasse, S., Nierstrasz, O. and Black, A.: Traits: Composable Units of Behaviour, *ECOOP 2003 - Object-Oriented Programming* (Cardelli, L., ed.), Lecture Notes in Computer Science, Vol. 2743, Springer Berlin Heidelberg, pp. 248–274 (online), DOI: 10.1007/978-3-540-45070-2_12 (2003).

[10] 竹下若菜，千葉 滋：破壊的クラス拡張で生じるメソッド衝突を回避可能なモジュール機構 Method Shells とその実装方法，情報処理学会論文誌. プログラミング，Vol. 7, No. 3, pp. 12–21（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110009806546/⟩ (2014).

[11] Wadler, P. and Blott, S.: How to Make Ad-hoc Polymorphism Less Ad Hoc, *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, New York, NY, USA, ACM, pp. 60–76 (online), DOI: 10.1145/75277.75283 (1989).

付　　録

A.1 PEG パーザコンビネータライブラリのソースコード

```
module PEG:
  class Expr:
    def MkExpr(proc)
  end
  class Result:
    def Success(pos, value)
    def Failure()
  end
  def parse(MkExpr(expr), str):
    expr(str, 0, Hash::create(16))
  end
  trait New():
    let nonterms = Hash::create(16)
    def def_nt(name, MkExpr(expr)):
      nonterms[name] = expr
    end
    def char(c):
      MkExpr^(str, pos, caches):
        if(pos < String::length(str) && str[pos] == c):
          Success(pos + 1, c)
        else:
          Failure()
        end
      end
    end
    def string(s):
      let len = String::length(s)
      MkExpr^(str, pos, caches):
        if(pos + len <= String::length(str) && String::sub(str, pos, len) == s):
          Success(pos + len, s)
        else:
          Failure()
        end
      end
    end
    def char_set(cs):
      MkExpr^(str, pos, caches):
        if(pos < String::length(str) && String::contain?(cs, str[pos])):
          Success(pos + 1, str[pos])
        else:
          Failure()
        end
      end
    end
    def nt_ref(name):
      MkExpr^(str, pos, caches):
        match(Hash::get(caches, (name, pos))):
        case(Some(result)):
          result
        case(None()):
          let result = nonterms[name](str, pos, caches)
          caches[(name, pos)] = result
          result
        end
      end
    end
    let fail = MkExpr^(str, pos, caches):
      Failure()
    end
    let no_op = MkExpr^(str, pos, caches):
      Success(pos, ())
    end
    def seq(MkExpr(expr1), MkExpr(expr2)):
      MkExpr^(str, pos, caches):
        match(expr1(str, pos, caches)):
        case(Success(pos, value)):
          expr2(str, pos, caches)
```

```
        case(Failure()):
          Failure()
        end
      end
    end
    def alt(MkExpr(expr1), MkExpr(expr2)):
      MkExpr^(str, pos, caches):
        match(expr1(str, pos, caches)):
        case(Success(pos, value)):
          Success(pos, value)
        case(Failure()):
          expr2(str, pos, caches)
        end
      end
    end
    def zero_or_more(MkExpr(expr)):
      MkExpr^(str, pos, caches):
        def loop(rev_values, pos):
          match(expr(str, pos, caches)):
          case(Success(pos, value)):
            loop([value, *rev_values], pos)
          case(Failure()):
            Success(pos, List::rev(rev_values))
          end
        end
        loop([], pos)
      end
    end
    def not(MkExpr(expr)):
      MkExpr^(str, pos, caches):
        match(expr(str, pos, caches)):
        case(Success(pos, value)):
          Failure()
        case(Failure()):
          Success(pos, ())
        end
      end
    end
    def return(value):
      MkExpr^(str, pos, caches):
        Success(pos, value)
      end
    end
    def bind(MkExpr(expr), proc):
      MkExpr^(str, pos, caches):
        match(expr(str, pos, caches)):
        case(Success(pos, value)):
          let MkExpr(expr) = proc(value)
          expr(str, pos, caches)
        case(Failure()):
          Failure()
        end
      end
    end
    def one_or_more(expr):
      bind(expr)^(value):
        bind(zero_or_more(expr))^(values):
          return([value, *values])
        end
      end
    end
    def zero_or_one(expr):
      alt(expr, no_op)
    end
    def and(expr):
      not(not(expr))
    end
    let String::C#(<-) = def_nt
    let Expr#(&+) = seq
    let Expr#(|+) = alt
    export def_nt, char, string, char_set, nt_ref, fail, no_op
    export seq, alt, zero_or_more, not
    export return, bind, one_or_more, zero_or_one, and
    export String::C#(<-), Expr#(&+), Expr#(|+)
  end
  export Success, Failure, parse, New
end
```