挿抜可能メソッドに基づく統一的な抽象単位複合機構

久野 靖[1,a)]

概要：今日のオブジェクト指向言語では、抽象単位 (クラス、Traits、アスペクト等) を組み合わせるのに、継承、型パラメタの束縛、AOP など多様な複合機構が使われている。しかしこれらは、最終的には適切なプロトコルを持つオブジェクトを構成するという点では共通している。1 つの言語が複数の複合機構を備えるということは、直交性の観点からは望ましくないし、各場面でどの機構を選択するかという問題が生じる。本発表では、統一的な抽象単位複合機構を持つ言語を提案する。提案機構は、ドメイン特化言語により抽象単位が持つメソッドの抜き出しや組み合立てを型安全な形で指示し、これを翻訳時に実行するものである。メソッド中の型名や非局所変数名は、メソッドを元の文脈から抜き出す際にパラメタに変換され、メソッドを他の文脈に入れる際に再度結合される。本機構は、既存の複合機構の主要部分を実装でき、また全く新たな複合機構の実装にも用いることができる。

キーワード：抽象単位, 複合機構, 挿抜可能なメソッド, 強い型

Unified Composition Mechanism for Abstraction Units Based on Pluggable Methods

Yasushi Kuno[1,a)]

Abstract: In today's object-oriented programming languages, abstraction units (such as classes, traits, aspects) are modified through various composition mechanisms; inheritance, parameter type binding and advice weaving (for aspects) are representative ones. However, those composition mechanisms are all the same in that they compose objects which implement desirable protocols. Having multiple compsition mechanisms defeats design orthogonality of a language, and raises (miss-)selection problems. In this presentation, we propose a programming language with an unified compsition mechanism. In this mechanisim, methods are individualy copied and combined according to dedicated domain-specific language description executed at comile time, in a type-safe manner. In this language, type names and non-local variable names within a method are converted to parameters when the method was "unplugged" from its original context. Those parameters can be rebound to actual types and variables when the method is "plugged" to another context. This mechanism can be used to implement large part of existing composition mechanism listed above, and also some completely new ones.

Keywords: abstraction entity, composiiton mechanism, pluggable methods, strong types

1. はじめに

今日の手続き型 (ないし命令型) プログラミング言語では、オブジェクト指向機能を持つもの (オブジェクト指向言語) が主流の 1 つとなっている。オブジェクト指向機能の重要な用途の 1 つは、データ抽象 (data abstraction) の機能を提供することである。これは具体的には、(1) データ (インスタンス変数群) とそれに対する操作 (メソッド) 群をまとまった単位 (オブジェクト) として定義し、(2) データをメソッドを通じてのみアクセス可能とすることで、オブジェクトの内部構造を外部の (オブジェクトを利用する) コードから隠蔽することを意味する。

多くのオブジェクト指向言語ではこのデータ抽象の単位をクラス (class) と呼んでいるが、本稿では特定の言語の性質に依存しないため抽象単位 (abstraction unit) と呼ぶこ

[1] 筑波大学ビジネスサイエンス系
Faculty of Business Sciences, University of Tsukuba
[a)] kuno@gssm.otsuka.tsukuba.ac.jp

とにする。

ソフトウェアの開発においてデータ抽象の機能を活用することは、次の利点をもたらす。

- 抽象単位の内部と外部の相互依存が小さく、それぞれを独立して開発できる。
- ある程度まとまった機能を抽象単位としてライブラリ化することで再利用が促進でき開発効率が高められる。

これらの利点を享受するためには、いちど完成した抽象単位は「そのまま」使える (使用箇所に応じて手直ししないで済む) ことが前提となる。これは、手直しすることになると、外部との相互依存が生じてしまうためである。

とはいっても、使用箇所ごとのニーズは極めて多様であるため、何らかの形でニーズに合わせるためのカスタマイズを施したい場合はある。このとき、完成している抽象単位の内部に直接手を入れるのではなく、外部から必要な修正部分を付加する形で行えれば、完成している部分を壊すことがなく、再利用の利点を失わないで済む。

この、付加部分も (それ自体では完結していないかも知れないが) 一種の抽象単位であると見なすなら、このようなカスタマイズ作業は複数の抽象単位を複合 (composition) させることで実現されると言える。

今日のオブジェクト指向言語において広く認められている複合機構としては、継承、パラメタ化、AOP が代表的である。歴史的には、これらの中では継承が最も古く、それでは賄えない部分をカバーするという形でパラメタ化や AOP などが追加されてきたという経緯がある。

しかし、1 つの言語上に複数の複合機構があることは、言語を複雑にし、また使い分けの問題をもたらす。

本稿ではこのような考えに基づき、静的な型を持つ言語において、継承、パラメタ化、AOP を 1 つの複合機構で取り扱う枠組みを探究する。ただし、AOP の中には動的な条件に基づく複合を含む部分があるが、本稿ではコンパイル時に動作する静的な複合機構を前提とするので、動的な条件については扱わない。

以下第 2 節では、代表的な複合機構である継承、パラメタ化、AOP を取り上げ、それぞれの機能を整理する。続いて第 3 節ではこれらの機能が共通して提供する機能について議論し、第 4 章ではそれに基づいて統一的・汎用的な複合機構の提案を行う。第 5 章では先行研究について紹介し、第 6 章でまとめをおこなう。

2. 代表的な複合機構

2.1 継承

本節では記述を簡潔にするため、抽象単位を「クラス」と記す。継承 (inheritance) とは、あるクラス (親クラス) を土台としてそこからの差分を記述することで新たなクラス (サブクラス/子クラス) を定義する機能であり、Simula、Smalltalk、C++、Java など多くのオブジェクト指向言語に古くから見られる。

言語によって詳細は異なるが、共通して見られる継承の機能 (サブクラス定義時に記述可能な差分) をまとめると次のようになる。

- クラスが定義するオブジェクトが持つインスタンス変数を新たに追加する。
- オブジェクトが持つメソッドを新たに追加する。
- メソッドの一部を新たな実装で差し替える。言語によってはメソッドの前後に新たなコードを追加できるものもある。

これにより、少ない記述で多数の関連するクラスを作成できる (差分プログラミング)。

一方、差分プログラミングにはクラス間の依存性を高め保守しにくいシステムを生み出すという批判もある。これに対処する方向の 1 つとして、特定のクラスに対する差分ではなく、一般的に適用可能な機能をクラスとして用意し (Mixin クラスや Traits などと呼ばれる)、多重継承 (複数の親クラスからの継承) などの機構により合成することで有用なクラスを作り出す方法がある。

また、静的な型を持つオブジェクト指向言語では、(パラメタを持たない) クラスが 1 つの型に対応するものが主流であるが、これらの言語では親クラスの型 P は子クラス C に対応する型を包含する (下位互換となる — 子クラスのインスタンスが親クラスのインスタンスとして振る舞うことができ、親クラス型の変数に子クラスのインスタンスを格納できる)。本稿ではこれを $C \leq P$ と記す。

これにより、ある変数に対するメソッド呼び出しは、実行時点でその変数に保持されているオブジェクトのクラスが持つメソッド呼び出しとなり、オブジェクト指向言語の特徴であるメソッドの動的分配 (dynamic dispatch) が実現される。

ただし、このような形で実装の差分記述と型に関する互換性が一体となっていることに対しては批判もあり、これらを分けようとする動きもある。Java のインタフェース機能はその一例である。

2.2 パラメタ化

パラメタという用語は個々の操作 (メソッド、手続き) に渡される引数を意味することもあるが、ここでいうパラメタ化 (parametrization) は、抽象単位がパラメタを持つことを指す。パラメタを型に限定している言語 (Java、Scala) と、組み込み型の値なども許すもの (CLU[11]、C++) とがある。

もともとこのような機構は、静的な型のある言語において、型のためだけに内容が同じ抽象単位 (または単独の操

作) を重複して記述することを避けたいという自然な要求に依っている。逆の面から見れば、抽象単位の実装中で特定の型 (や値) に決め打ちする必要がないものはすべてパラメタとすることにより、抽象度や再利用性の高い実装を記述する、というのがパラメタ化の目的だと言える。

パラメタ化の典型例は配列のような単なるコンテナ (入れもの) を定義する抽象単位である。配列などでは、特定の型の値を単に保持するだけでその型の操作を呼び出すことはないので、パラメタ型に対して要件が課されることはない。

しかし要素の大小比較を必要とする順序集合のように、パラメタ型に対する要件が必要なものもあり、その要件の指定方法については言語によって次のような複数の流儀がある。

- 要素が持つべき操作のシグニチャを列挙 — CLU など。
- 要素が実装するインタフェースや属するクラス階層上の位置などを指定 — Java、Scala など。
- パラメタを埋め込んでコンパイルした結果の成否による — C++。
- 要素が持つべき要件を規定するモジュール様の機構の提供 — C++コミュニティで検討されている concepts[7] など。

また、単なる要件にとどまらず、パラメタとして指定された型が何であるかによって枝分かれして複数の実装を切り替える場合もある (C++におけるテンプレート特殊化など)。

型以外の (整数など組み込み型の値の) パラメタを持つ言語の場合 (もっぱら C++)、再帰的な展開を通じてコンパイル時に複雑なコードを合成することも可能である。このような技法はテンプレートメタプログラミングとして知られている [5]。

パラメタを持つ抽象単位の場合、パラメタをすべて指定した状態で完結した型定義となる。このため、パラメタを持つ抽象単位のことを型の観点からは型生成子 (type generator) と呼ぶこともある。型生成子により定義される型の包含関係については、言語によりさまざまな流儀がある。たとえば Scala はパラメタ群が値の書き込み/取り出しを許すかどうか、実装内部でどのように使用されているかに応じて包含関係の有無や方向を決定するようになっている。

Scala は型パラメタ機構がある一方、継承でも型パラメタと同等の機能が実現できる [12]。具体的には次のように、抽象クラスの要素として型を含め、子クラスにおいてその型を具体的な型で上書きすればよい。

```
abstract class AbsCell {
  type T
  val init: T
  private var value: T = init
  def get: T = value
  def set(x: T):unit = { value = x }
}
...
val cell = new AbsCell {
  type T = int; val init = 1
}
```

しかし、同じことを行うのに複数の手段があることが言語として望ましいかは、議論のあるところである。

2.3 AOP

AOP (Aspect Oriented Programming) とは、オブジェクト指向が提供する構造に対して横断的な関心事 (croscutting concern) をそれ単独に記述し実装する考え方を意味し、その独立して記述される関心事をアスペクトと呼ぶ。

オブジェクト指向の構造に沿って素直に記述できない横断的関心事 (アスペクトとして実装するのが適する機能) の例としては、並列実行単位間の同期、ログの記録、画面の再描画などがしばしば挙げられる。

AOP のための言語/言語機構としては、AspectJ[8]、SOP (Subject Oriented Programming[13])、Composition Filter[1]、Demeter/Adaptive Programming[10] などがある。これらが提供する機能を簡潔にまとめると、(1) プログラム実行における特定の箇所 (コード上の位置+時点) を指定し、(2) その箇所で実行すべき動作を指定し、さらに必要なら (3) 前記を実現するための変数やメソッドの追加を指定するものである。そして 1 つの横断的関心事に関する (1)〜(3) をまとめたものが 1 つのアスペクトとなる。

ここで、コード上の位置はおおむね、メソッドの入口や出口 (いずれも呼び側・呼ばれ側の双方があり得る) をメソッド名 (パターンを用いる場合もある) などの形で指定するが、それに加えて「時点」として特定のメソッドの実行中など動的な (実行時の) 条件を指定できるものがある (前述したように、本稿では動的な条件については扱わない)。

3. 統一的な複合機構

3.1 複合機構統一に向けての検討

本節では抽象単位の統一的な複合機構に関する検討をおこなう。その前提として、新たな複合機構は既存の複合機構にとって代わるものであり、既存の複合機構で可能なことは (静的なものの範囲内で)、できるだけ多くカバーしたい。このために、前節までで挙げた複合機構において、それぞれ既存の抽象単位に対してどのような操作が行われているかを整理する。

まず継承によって行われる操作は、親クラスと子クラスをそれぞれ別の単位として考え、これらに基づいて新たな

単位が作られるものとすると、次のようにまとめられる。

- 新単位が持つインスタンス変数群は親単位のものに子単位のものを追加したものである。
- 新単位が持つメソッド群は親単位のものに子単位のものを追加したものである。
- 一部のメソッドについては、子単位のものが親単位の実装を上書きするか、あるいは親単位のものの前や後に付加されて実行されるようになる。
- 新単位が定義する型は親単位の型に包含される。

次にパラメタ化については、次のようにまとめられる。

- 抽象単位が参照する名前 (値、型) の一部がパラメタとして宣言されており、パラメタに対して具体的な値や型を束縛することで単位が機能する (具体化)。
- 単位が公開するインタフェースは、そこに現れるパラメタを単に置換するなどの形で、またパラメタとして指定された値や型が持つインタフェースの全体または一部を流用するなどの形で、パラメタに依存することがある。
- 単位が定義する型の型階層上の位置づけは、パラメタに依存しない部分もあり得るし、また (共変的に、ないし反変的に) 依存する部分もあり得る。

最後に AOP については、次のようにまとめられる。

- アスペクトに相当する単位は、それが修飾する単位について、どの箇所をどのように修飾するかの記述を含む。
- 箇所については、指定されたメソッドの出口や入口 (呼び側の場合も呼ばれ側の場合もある) が一般的である。
- 修飾の方法については、指定位置において実行する追加動作をメソッドの形で指定するのが一般的である。
- これらに加えて、修飾される単位側に新たなインスタンス変数やメソッドを追加する場合もある (インタータイプ宣言)。

3.2 基本的なアイデア

ここまでに整理したことがらをまとめると、さまざまな複合機構はいずれも、(1) 型に対する操作と (2) 実装に対する操作が組み合わさってできている。本稿の提案は、これらの操作をコンパイル時に実行されるドメイン特化言語 (DSL) によって記述可能とすること、である。

まずその前提として、既定の抽象単位を、従来のクラスと同様に定義しておく。抽象単位は全く実装を持たないインタフェースである場合と、インタフェースに加えてインスタンス変数群やメソッド定義を含んだものである場合がある。そして、DSL ではこれらを「素材」として新たな型や抽象単位を組み立てる。

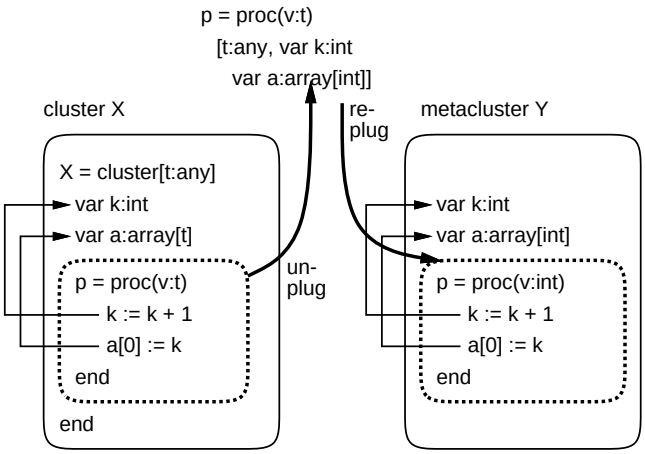

図 1　メソッドの挿抜

以下では (1)、(2) それぞれに分けて、考え方を記す。

型に対する操作

型についてはさまざまな定義の流儀があるが、本稿では抽象データ型の考えに基づき、型とは「メソッドのシグニチャ (名前ならびパラメタの個数と型、返値の個数と型) の集まり」であるものとし、さらに多くのオブジェクト指向言語にならって「型どうしは包含関係を持つ (型階層を構成する)」ものとする。この前提では、型に対する操作としては、次のものがある。

(a)　その型に属するシグニチャの集合を定める。

(b)　その型が属する型階層上の位置を定める。

(a) については、ある型を新規に定義する際、最初はシグニチャを持たないものとし、そこに他の型が持つシグニチャを選択的に、必要なら一部を変更しつつ追加する操作を DSL で記述する。

(b) については、新たな型を定義する際に、その型がどの型と互換かを DSL の記述で明示するものとする。

任意に包含関係を規定した場合、それらの型が実際には (必要なメソッドが欠けているなどの形で) 互換でないこともあり得るが、これは DSL の実行時エラーとする (コンパイル時 DSL なので、実際にはこのようなエラーはコンパイル時に発生する)。

実装に対する操作

抽象単位はインスタンス変数群とメソッド群を持ち、インタフェースに規定された (外部からアクセス可能な) メソッドが呼び出されたとき、そのメソッド内に含まれる式・文が評価・実行されることを通じて動作する (その過程でインスタンス変数群が読み書きされる)。

既存の継承やアスペクト指向においては、メソッド単位での差し替えや付加によって実装の拡張を行っていること

を考慮し、本提案でもメソッド内部の修正は行わないものとした。そのため、DSL による実装の操作は次の形で動作するものとする (図 1)。

(c) 既存の抽象単位からメソッドを取り出し、構築中の抽象単位に追加する。この場合、既存のものを置き換える場合と、既存のメソッドの直前/直後に実行されるように付加する場合とがある。

このため、抽象単位に属する各メソッドは、その定義された文脈から抜き出して他の抽象単位に差し込むことができる (pluggable、挿抜可能) ものとする。抜き出す際に、元の文脈に存在するインスタンス変数への参照や型名への参照はパラメタに変換され、新たな文脈に差し込まれた時に再束縛される。

すなわち、本提案ではパラメタ機構は言語の基本機構として組み込まれており、その上で DSL を通じて継承やアスペクトなどの複合機構を記述することになる。このような選択は、λ 計算をはじめとする多くの計算モデルにおいて名前の置き換えが基本機構として組み込まれていることなどからも、不自然ではないと考える。

4. o3: 実証用オブジェクト指向言語

4.1 o3 の設計方針と基本構造

前節で述べた提案の実現性・実用性を評価するため、実証用言語 o3 を設計し実装した。その設計方針は次の通り。

- 提案機構に関わらない基本部分は普通のオブジェクト指向言語とする。
- 提案機構に関わらない部分はできるだけ簡潔なものとする。
- 提案機構に関わる部分は他の部分から明確に分離する。

o3 の基本部分の構文を簡略化したものを図 2 に示す。プログラムはモジュールの並びであり、モジュールにはインタフェース、クラスタ (他の言語のクラスに相当)、メタ手続き、メタクラスタの 4 種類がある。

インタフェースは型 (メソッドシグニチャの集まり) を定め、クラスタは型とその実装 (インスタンス変数群およびメソッドの実装) を定める。クラスという名称を用いなかったのは、継承の機能を持たないためである (継承をはじめとする複合機構は提案機構により実現)。

前述のように、型パラメタは基本的な機構としたので、インタフェースやクラスタも、型パラメタを持つことができる。メタ手続きはその性質上、型パラメタが不可欠である。メタクラスタについては、型パラメタの有無は通常のクラスタの場合と同じ意味になり、型パラメタがある場合は型生成子を定義する。

メタ手続きおよびメタクラスタは型・クラスタ組み立て

```
program ::= ( interface | cluster | metadef )...
interface ::= idn = interface [ param ] annot...
    procdcl... end
cluster ::= idn = cluster [ param ] annot...
    vardef... procdef... end
annot ::= @idn [ [ ( idn | string | integer )... ] ]
param ::= [ ( idn : type )... ]
type ::= idn [ [ type,... ] ]
prochdr ::= proc ( ( idn : type )... ) [ : type ]
procdcl ::= idn = prochdr end
procdef ::= idn = prochdr stat... end
vardef ::= var idn : type [ := expr ]
stat ::= vardef | assign | astore | rstore | simpcall
    | return [ expr ] | whilest | ifst
whilest ::= while expr do stat... end
ifst ::= if expr then stat... [ elif expr then stat... ]...
    else stat... end
assign ::= idn := expr
astore ::= idn [ expr ] := expr
rstore ::= idn . idn := expr
expr ::= simcall | uop expr | expr bop expr | ( expr )
uop ::= + | - | !
bop ::= = | != | > | >= | < | <= | + | - | * | / | %
    | && | ||
simpcall ::= ( primary | simpcall ) ! idn( expr,... )
  | $type$ idn ( expr,... )
primary ::= idn | integer | string | true | false | nil
    | primary [ expr ] | primary . idn
```

... — 0 or more repetiotion
,... — comma-separated list
[...] — optional

図 2 o3 の構文 (簡略化したもの)
(文の末尾には; を置くが、すべて省略可能)

DSL(以下では単に DSL と記す) を記述するモジュールであり、その記述内容はコンパイル時に解釈実行される (詳細は後述)。

4.2 o3 の基本部分

本節では o3 の基本 (オブジェクト指向言語) 部分について簡単に解説する。メソッド呼び出しは「$obj!method(\cdots)$」または「$\$type\$method(\cdots)$」の形で記述する。前者は動的分配を行う通常の呼び出しであり、後者はインスタンスの生成などのためにクラスタ等を直接指定した呼び出しとなる。

図 3 に組み込みインタフェース/クラスタの定義を示す。any のみがインタフェースであり、「すべての型を包含する型」として扱われる。bool、int、string はそれぞれ論理値、整数、文字列のオブジェクトであり、リテラルとして記述できる。また論理値は if や while の条件部分に表われこれらの実行を制御する。これらのことから、この 3 つを組み込みとしている。また、array は配列であり最も基本

```
any = interface end
bool = cluster
  equal = proc(self:bool, x:bool):bool end
  not = proc(self:bool):bool end
  print = proc(self:bool) end
end
int = cluster
  equal = proc(self:int, x:int):bool end
  lt = proc(self:int, x:int):bool end
  gt = proc(self:int, x:int):bool end
  le = proc(self:int, x:int):bool end
  ge = proc(self:int, x:int):bool end
  minus = proc(self:int):int end
  plus = proc(self:int):int end
  add = proc(self:int, x:int):int end
  sub = proc(self:int, x:int):int end
  mul = proc(self:int, x:int):int end
  div = proc(self:int, x:int):int end
  mod = proc(self:int, x:int):int end
  print = proc(self:int) end
end
string = cluster
  equal = proc(self:string, x:string):bool end
  lt = proc(self:string, x:string):bool end
  gt = proc(self:string, x:string):bool end
  le = proc(self:string, x:string):bool end
  ge = proc(self:string, x:string):bool end
  add = proc(self:string, x:string):string end
  size = proc(self:string):int end
  print = proc(self:string) end
end
array = cluster[elt:any]
  create = proc():array[elt] end
  size = proc(self:array[elt]):int end
  push = proc(self:array[elt], x:elt) end
  store = proc(self:array[elt], i:int, x:elt) end
  fetch = proc(self:array[elt], i:int):elt end
end
```

図 3 o3 の組み込みクラスタ定義

的なコンテナとして組み込みとした。このクラスタは型パラメタを持ち、そのパラメタ型の値を格納する。

ここで o3 の型階層について説明する (図 5)。まず、すべての型 T は any 型に包含される ($T \leq$ any)。これは、任意の型を受け取るパラメタなどの指定に必要なためそのようにした。これ以外の型どうしの関係は全て DSL によって明示的に定める。

1 つの型に対し複数の親の型を指定できるので、$\leq$ は半順序をなし、any がその最大元となる。このため、任意の型 A、B に対し、共通の上界は常に存在する (その極小のものは複数あるかも知れない)。一方、任意の型 A、B に対し、共通の下界が存在するかどうかは場合による。

o3 の簡単なプログラム例を図 4 に示す。配列をもとにスタックを定義し、main ではスタックを生成して値をいくつか積み、取り出して出力している。クラスタの中では create というメソッド名は特別であり、変数 self が定義され、そこに新たに作られたオブジェクトが格納された状態で実行を開始する。

```
stack = cluster[elt:any]
  var arr:array[elt] := $array[elt]$create()
  var ptr:int := 0
  create = proc():stack[elt] return self  end
  push = proc(self:stack[elt], x:elt)
    if arr!size() > ptr then
      arr[ptr] := x; ptr := ptr + 1
    else
      arr!add(x); ptr := ptr + 1
    end
  end
  pop = proc(self:stack[elt]):elt
    if ptr >= 0 then ptr := ptr - 1 end
    return arr[ptr]
  end
  isempty = proc(self:stack[elt]):bool
    return ptr <= 0
  end
end

test = cluster
  main = proc()
    var st:stack[int] := $stack[int]$create()
    st!push(1); st!push(2); st!push(3)
    st!pop()!print(); st!pop()!print()
  end
end
```

図 4 簡単な例題 (スタック)

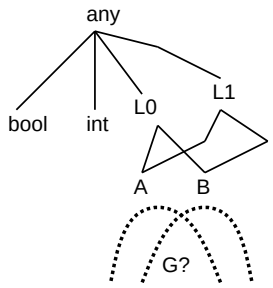

図 5 o3 の型階層

4.3 型・クラスタ組み立て DSL

型・クラスタ組み立て DSL は前述のように、メタ手続き・メタクラスタとして記述する。メタ手続きとメタクラスタは構文は同一であるが、メタクラスタが実際のクラスタを組み立てるのに対し、メタ手続きはメタクラスタからパラメタとともに呼び出されて一連の手順を実行する。す

metadef ::= *idn* = (metaproc | metacluster)
[*param*] *annot*... *mstat*... end
mstat ::= *mcall* | *mfor* | *mif*
mfor ::= for *idn*: *mexp* do *mstat*... end
mif ::= if *mexp* then *mstat*... [elif *mexp*
then *mstat*...]... else *mstat*... end
mexp ::= *mcall* | *type* | $*type*$*idn* | *string*
mcall ::= *mexp*!*idn*[*mexp*...]

図 6 型・抽象単位組み立て DSL の構文 (簡略版)

表 1 DSL で提供されるオブジェクトと API(抜粋)

type	
add_proctype[*proc*]	Add *proc*'s signature to target
add_proctypes[*type*]	Add *type*'s all procs' signatures to target
add_super[*type*]	Add *type* as target's supertype
proctypes[]	Retuns target's set of *proc*
cluster	
add_procdef[*proc*]	Add *proc*'s body to target
add_procdefs[*cluster*]	Add *cluster*'s all procbodies to target
procdefs[]	Retuns target's set of *proc* with body
proc	
add_body_after[*proc*]	Append *proc*'s body to target
add_body_before[*proc*]	Prepend *proc*'s body to target
name_matches[*string*]	Test if *proc*'s name matches pat

なわち、メタ手続きの目的は、一連の DSL 記述に名前をつけて共有したり構造化 (抽象化) することである。

DSL の本体では、図 6 にあるように、メタ文によって動作を記述する。単純な文は o3 の基本部分と類似したメッセージ送信式であり、各動作は予め定められた API によって提供される (表 1)。

メソッドを既定クラスタから抜き出したとき、メソッドがそのクラスタのイスタンス変数を参照している場合は変数パラメタに変換される (変数パラメタを直接指示する機能は無い)。その後、メソッドを他のクラスタに付加したときに、そのクラスタにインスタンス変数が (無ければ) 追加され、変数パラメタはそこに束縛される。従って、同名の変数で型が違うものを参照しているメソッドを 1 つのクラスタに付加しようとするとエラーになる。[*1]

オブジェクトとしては、型 (シグニチャの集合であり、親クラスの集合を持つ)、クラスタ (型に加えてメソッド定義の集合を持つ)、メソッド (シグニチャとあれば本体を持つ) が主要なものである。このほか、パターンや名前を指定するための文字列オブジェクト、述語の結果を返すための論理値オブジェクトがある。オブジェクトの型は実行時 (o3 全体として見ればコンパイル時) に動的検査される。このほか制御構造として、条件により枝分かれする if 文と集合の各要素に対して反復する for 文のみを用意している。

[*1] このほか、型が違うものごとに別個の変数とすることや、差し込む際に名前替えを指示することも、今後検討したい。

```
accum = cluster
  var value:int
  create = proc():accum value := 0; return self end
  inc = proc(self:accum, n:int) value := value + n end
  get = proc(self:accum):int return value end
end

exaccumimpl = cluster
  var value:int
  reset = proc(self:accum) value := 0 end
end

extends = metaproc[target:any, parent:any, child:any]
  target!add_super[parent]
  target!add_proctypes[parent]
  target!add_proctypes[child]
  target!add_procdefs[parent]
  target!add_procdefs[chlid]
end

countimpl = cluster
  var count:int := 0
  create = proc():countimpl return self end
  countup = proc(self:countimpl) count := count + 1 end
  getcount = proc(self:countimpl):int return count end
end

addcount = metaproc[target: any]
  $target$create!add_body_after[$countimpl$create]
  target!add_proctype[$countimpl$getcount]
  target!add_procdef[$countimpl$getcount]
  for p: target!procdefs[] do
    if p!name_matches["^(inc|reset)"] then
      p!add_body_after[$countimpl$countup]
    end
  end
end

exaccum = metacluster
  selftype!extends[accum, exaccumimpl]
  selftype!addcount[]
end
```

図 7 継承と記録の追加を行う例題

4.4 例題: 継承と記録の追加

本節では継承とアスペクトに相当する機能をメタ手続きとして作成する例を示す (図 7)。拡張対象となるクラス accum は整数の累計を保持するオブジェクトを定義し、生成のためのメソッド create、値を増加するメソッド inc、現在を読み出すメソッド get を定義している。ここで新たにメソッド reset を持つように拡張したクラスタを定義したいとする。このため reset の実装を持つクラスタ

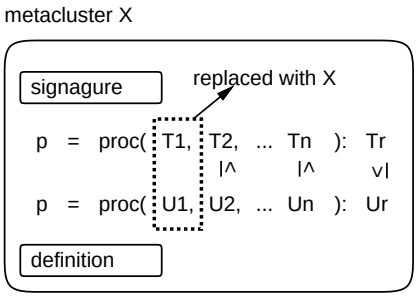

図 8　シグニチャとメソッド定義の互換性

exaccumimpl を用意した。

このような拡張を行うメタ手続きが extends[] であり、型 target に parent と child のメソッドを追加することで拡張をおこなう。その中では、まず target の親クラスとして parent を追加し、次に target に parent のメソッドシグニチャと child のメソッドシクニチャを追加する。続いて、parent が持つメソッド実装と child が持つメソッド実装を target にコピーする。ここでは修正や選択は行わないので、すべてのシグニチャや実装をコピーしているが、必要なら for 文で 1 つずつ列挙しながら修正や選択を行うこともできる。

次に、変更を行うメソッドが呼び出されるごとにその回数を記録する処理をアスペクトのように差し込むことを考える。記録する動作の実装はクラスタ countimpl として用意した。

実際に差し込みを行うメタ手続き addcount は、まず対象クラスタの create の実装の後にカウンタを初期化する $countimpl$create の実装を付加する。次に、カウント値を取得するメソッド getcount のシグニチャと実装を追加する。最後に、対象クラスタが持つすべてのメソッドについて調べ、変更を行うメソッド (ここでは名前で選択) について、その実装の末尾に countup の実装を付加する。

実際の拡張クラスタはメタクラスタ exaccum で定義されており、その中では継承のための extends と記録動作付加のための addcount を呼び出している。メタクラスタ中では名前 selftype がそのメタクラスタ自身が定義している型およびクラスタを表す。

4.5　DSL の実行時処理と制約

DSL では型の親子関係や型が持つシグニチャ、クラスタが持つメソッド定義をかなり自由に操作できるが、前述のように、結果として構築された型やクラスタに矛盾がある場合は実行時エラー (o3 処理系全体としてはコンパイル時エラー) とする。具体的に実行される処理や満たすべき制約としては、次のものがある。

- 型の親子関係にループが存在してはならない。
- シグニチャを付加する際、その第 1 引数の型が対象クラスタの型を包含するなら、第 1 引数の型は対象クラスタの型で置き換える。[*2]
- 同名のシグニチャを複数回追加する際、引数の個数および返値の有無は一致する必要があり、(前項の場合を除き) 引数の型は当該位置のパラメタ型すべての共通の上界の最小の型、返値の型はすべての返値型の共通の下界の最大のものとする (最小/最大が一意でなかったり後者で共通の下界が無い場合はエラーとする)。
- 実装を複数回追加する場合、最後に追加された実装が有効となる (上書き)。
- 追加されるメソッド実装は、対応するシグニチャに互換でなければならない。互換とは、各引数の型はシグニチャの対応する引数の型を包含し、返値の型はシグニチャの返値に包含される必要がある (図 8)。[*3]
- 本体メソッドの前や後にメソッドを付加する場合、その引数の個数は本体メソッドの引数の以下であり、各引数の型は上と同様の制約があるものとする。返値の有無は任意 (返値は無視される)。[*4]

表 2　o3 処理系の実装規模

	#. of lines
SableCC grammer	255
Java code	2000

5.　o3 の実装

o3 の実装は構文解析に SableCC[6] コンパイラコンパイラを使用し、Java 言語により実装している (表 2)。内部では抽象構文木を組み立てた後、通常のインタフェース・クラスタは意味解析に基づき型テーブルに型や実装の情報を構築し、型・クラスタ DSL についてはツリーインタプリタとして実行しながら型テーブル中の型や実装に変更を加えてから意味チェックを行う。最終的に完成した型テーブル中の構造をもとにコード生成をおこなう (図 9)。本処理系は検証用であり、分割コンパイルなどの機能は備えていない。

生成コードは C 言語であり、C コンパイラによって翻訳した後実行可能となる。オブジェクトはすべて先頭にメソッドテーブルへのポインタを持ち、その後にオブジェクトの種別に応じたインスタンス変数群の領域が含まれている。メソッドテーブルに配置されるのはメソッド群を束ねるベクタへの参照であり、このベクタは本体メソッドのみの場合は 1 個、前後にメソッドが付加されている場合には

[*2] 第 1 引数は動的分配の対象となるため。この置き換えを行わない選択肢 (API 呼び出し) も別途用意する。
[*3] この規則に合致しない場合に、引数や返値を変換するコードを差し込む機能を API に用意することも今後検討する。
[*4] これに加えて、後に付加するメソッドが本体メソッド (やそれに付加されたメソッド) の返値を受け取って加工し新たな返値を返す機能も今後検討する。

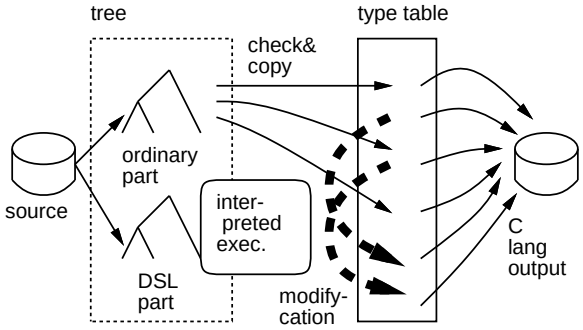

図 9　o3 処理系の構成

その個数だけの C 言語の関数ポインタを (それぞれの個数情報とともに) 格納している。

標準型の実装は o3 によるインタフェース記述と混在させたアノテーションに C 言語で生成すべきコードを記述してあり、一般のコードと一緒に実行コードが生成される。ガベージコレクションには保守的 GC[3] を使用している。

6. 関連研究

継承、型パラメタ (ジエネリクス)、AOP それぞれについては多くの研究がなされているが、これらを統合しようとする試みは多くない。

冒頭でも紹介したように、Scala[12] では抽象クラスを持つクラスの子クラスにおいて抽象クラスを具象クラスに上書きすることで型パラメタと同等の働きをさせられることを述べているが、これに型パラメタ機能を統合することは考えられていない。また、継承という機構自体が複雑な複数の機能から成るため、これをより小さい機能群に分解するというのが本稿の提案である。

Bergmans[2] らは継承と AOP を統合したクラス複合機構を持つ言語を提唱しているが、静的な型を持たない言語の枠組みにおいてであり、またメソッド呼び出し時に動的なマッチング機構による計算が介在することから、本稿で提案する静的な型検査も含めた枠組みとは異っている。

本稿で提案しているような継承時やメソッド呼び出し時の制御については、メタオブジェクトプロトコル (MOP) でも対応している (たとえば [9]、[4] など)。ただし、これらの枠組みは通常のオブジェクト指向を前提とした上で、継承時や呼び出し時の機能を拡張しており、継承を置き換えることは意図していない。

7. 議論とまとめ

本稿では継承、型パラメタ、AOP などに代表される多様な抽象単位複合機構を統合した言語について検討した。

提案する言語では、パラメタによる置き換えが土台の機構となり、既定義の抽象単位からシグニチャやメソッド実装を抜き出して新たな抽象単位に追加する形で抽象単位の複合 (composition) を行う。このため、基本となる言語に加えて抽象単位の構築を記述する DSL を内蔵し、処理系は DSL 部分をコンパイル時に解釈実行することにより抽象単位の組み立てを行う。

基本的なアイデアの実現性を評価するため、検証用のプログラミング言語 o3 を設計し実装した。実装に基づく第一の知見としては、このような言語が実際に作れ、それでプログラムを書くことも問題ない、ということである。

ただし、従来のオブジェクト指向では「extends 親クラス名」とだけ記せばよかった記述が繁雑になったことは否めない。その主要な理由は、メタクラスタと子クラスの実装部分にある既定クラスを分けて書かなければならないということがある。この部分をたとえばメタクラスタの末尾に一緒に書けるようにすれば、既存の言語と (継承の利用については) さほど違わなくなるように思える。

AOP に相当する記述については、(動的なものは除外した) 基本的な記述はあまり問題なく書けそうである。ただし、コードを差し込む部分の指定方法が現在のところメソッド名のパターンによるものに限られている。よりアドホックでない方法として、メソッドにアノテーションを付加し、そのアノテーションに基づく指定を行うことが考えられる。アノテーションをソースコードレベルで付加しておきたくなければ、メソッドを調べて適切にアノテーションを行うように DSL の API を強化することが考えられる。

この点も含め、DSL が持つ API については、今のところ最小限のものしか実装していない。有用な言語機構をメタ手続きとして記述するためには、DSL にどのような API が備わるべきかを、有用な言語機構の探究も含め、今後さらに検討して行きたい。

本研究は科学研究費助成事業 (研究課題番号: 25330076) によっています。

参考文献

[1] Mehmet Aksit, Anand Tripathi, Data abstraction mechanisms in SINA/ST, Proceedings of OOPSLA'88, pp. 267-275, 1988.

[2] Lodewijk Bergmans, Wilke Havinga, Mehmet Akist, First-Class Compositions — Definition and Composing Object and Aspect Compositions with First-Class Operators, Transactions on AOSD IX, Springer LNCS 7271, pp. 216-267, 2012.

[3] Hans-Juergen Boehm, Mark Weiser, Garbage collection in an uncooperative environment, Software: Practice and Experience, volume 18, issue 9, pp. 807-820, 1988. DOI: 10.1002/spe.4380180902

[4] A metaobject protocol for C++, Proceedings of OOPSLA'95, pp. 285-299, 1995. DOI: 10.1145/217838.217868

[5] Krzysztof Czarnecki, Ulrich W. Eisenecker, Generative Programming — Methods, Tools, and Applicaitons, Addison-Wesley, 2000.

[6] E. M. Gagnon, L. J. Hendren, SableCC, an object-oriented compiler framework, TOOLS 26 Proceedings, pp. 140-154, 1998. DOI: 10.1109/TOOLS.1998.711009

[7] Douglas Gregor, Jaakko Jarvi, Jeremi Siek, Bjane Stroustroup, Gabriel Dos Reis, Andrew Lumsdaine, Con-

cepts: Linguistic Support for Generic Programming in C++, OOPSLA'06, pp. 291-310, 2006.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold, An Overview of AspectJ, Proceedings of ECOOP 2001, Splinger LNCS 2072, pp. 327-354, 2001.

[9] Gregor Kiczales, Jim des Riviéres, Daniel G. Bobrow, The art of the metaobject protocol, MIT Press, 1991.

[10] Karl Lieberherr, Dough Orleans, Joan Ovlinger, Aspect-oriented programming with adaptive methods, CACM, volume 44, issue 10, pp. 39-41, 2001.

[11] Barbara Liskov, John Guttag, Abstraction and Specification in Program Development, MIT Press, 1986.

[12] Martin Odersky et. al., An Overview of the Scala Programming Language 2nd ed., Technical Report LAMP-REPORT-2006-001, EPFL Lausanne, Switzerland, 2006.

[13] Harold Ossher, Peri Tarr, Using subject-oriented programming to overcome common problems in object-oriented software development/evolution, Proceedings of ICSE'99, pp. 687-688, 1999.