プログラミング言語o3の言語設計と実装
Design and Implemantation of o3 Programming Language

Research Report No. 2016-01

久野　靖
Yasushi KUNO

February 2016

Graduate Schol of Systems Management
University of Tsukuba, Tokyo
3-29-1 Ootsuka, Bunkyo-ku, Tokyo 112-0012 JAPAN
email: kuno@gssm.otsuka.tsukuba.ac.jp

Abstract

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。プログラミング言語 o3 は、これらの課題に対処する方法として「統一的な抽象単位複合機構」を持つ言語を提案する目的で設計・開発しているのである。o3 は通常のコード記述に加え、抽象単位を操作するためのドメイン特化言語を持ち、これを仕様して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する。本レポートは Java 言語および SableCC コンパイラコンパイラを使用した o3 言語の実装について解説するとともに、o3 言語の設計の要点について議論している。

1　o3 言語の基本的な考え方

型を持つオブジェクト指向言語における抽象単位操作機構としては、継承 (Java, C++など多数)、型パラメタ (Java, Scala, ...) ないしテンプレート (C++)、アスペクト (AspectJ) など多くのものがあるが、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。o3 の基本的な提案は、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。

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。

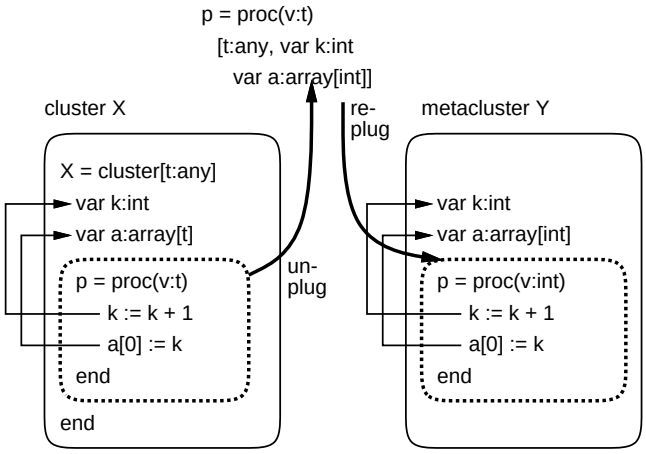

Figure 1: メソッドの挿抜

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。

2 o3 言語の設計

2.1 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 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)。

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

Figure 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)

```
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
```

Figure 3: o3 の組み込みクラスタ定義

```
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
```

Figure 4: 簡単な例題 (スタック)

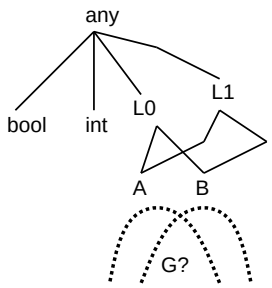

Figure 5: o3 の型階層

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...*]

Figure 6: 型・抽象単位組み立て DSL の構文 (簡略版)

2.2 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。

2.3 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

Table 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

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。

2.4 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 `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

[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
```

Figure 7: 継承と記録の追加を行う例題

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。

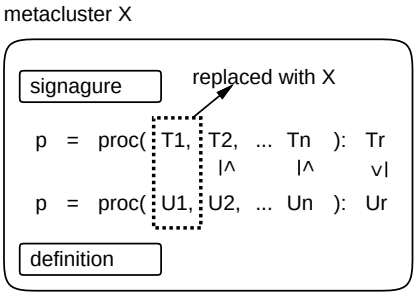

Figure 8: シグニチャとメソッド定義の互換性

2.5 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]

[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。

- 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]

Table 2: o3 処理系の実装規模

	#. of lines
SableCC grammer	255
Java code	2000

3 SableCC による字句・構文定義

o3 言語処理系の字句解析および構文解析の実装には、SableCC コンパイラコンパイラ [6] を使用している。以下に解説をおこない、末尾に SableCC への入力ソースを掲載する。

- `Tokens` — 言語上に現れるキーワードは字句解析レベルにおいて予約語として定義しており、通常の識別として使用することはできない。
- `prog` — プログラム上の単位としては通常単位とメタ単位の 2 種類がある。
- `annot` — 各単位やメソッドにはアノテーションがつけられる。アノテーションは「`@` 名前」ではじまり、それ単独で終わってもよいが、かっこで囲まれた中に名前、整数、文字列が複数並んでいてもよい。アノテーションは言語の特別な機能のために使うこともあるが、実装内部で標準機能実装のための他言語コードを埋め込むのに使用することもある。カンマやセミコロンは読みやすさのために書けるようになっているが、すべて省略可能である。
- `mclu` — メタ単位はメタクラスタとメタ手続きの 2 種類に分けられる。それらの中身はメタ文なので構造は同じだが、メタ手続きは共有されるメタ文の集まりをくくり出すのに使用する。
- `mstat` — メタ文はメタ呼び出し、メタ for 文、メタ if 文の 3 種類から成る。これらで十分かどうかは検討段階である。
- `mcall` — メタ呼び出しは DSL から o3 の単位操作機能を呼び出すインタフェースとなっている。現在は表 1 に記載されている機能がハードコードにより組み込まれている。
- `clus` — 通常単位はクラスタとインタフェースの 2 種類がある。インタフェースはメソッド実装を書かないだけなので構文としては同じである。いずれも型パラメタを持てる。
- `vdef` — 変数定義は初期化式を持つこともできる。
- `pdef` — メソッドは値を返すものと返さないものの 2 種類があり構文でも分けられている。
- `tname` — 型指定名前に加え実パラメタを持つものもある。

[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。

- fbody, vbody — 文以下は値を持つものと持たないものが並行して定義されている。文は代入/呼び出し、return、while、if の 4 種類になる。
- ascal — 代入/呼び出しは配列代入、フィールド代入を含むが、これらは実際にはメソッド呼び出しの構文糖となっている。
- calex — 呼び出しは「$型$名前 (...)」のものと「式!名前 (...)」のものがあり、後者がメソッド呼び出しで一般に使われる形となる。ただし構文上の制約から複雑な式が書けない位置がある。

```
Package o3;

Helpers
  all = [0..0xffff];
  digit = ['0'..'9'];
  lalpha = ['a'..'z'];
  ualpha = ['A'..'Z'];
  alpha = lalpha | ualpha;
  ascii = [0..0x007f];
  quote = '"';
  plus = '+';
  minus = '-';
  uscore = '_';
  sign = plus|minus;
  nostr = [ascii - quote];
  escape = '\' ascii;
  dot = '.';

Tokens
  cluster = 'cluster';
  do = 'do';
  else = 'else';
  elsif = 'elsif';
  end = 'end';
  false = 'false';
  for = 'for';
  if = 'if';
  interface = 'interface';
  metacluster = 'metacluster';
  metainterface = 'metainterface';
  metaproc = 'metaproc';
  nil = 'nil';
  var = 'var';
  proc = 'proc';
  return = 'return';
  then = 'then';
  true = 'true';
  while = 'while';
  atidn = '@' (alpha|digit)+ ;
  dollar = '$';
  plus = '+';
  minus = '-';
```

```
  mult = '*';
  div = '/';
  rem = '%';
  lpar = '(';
  rpar = ')';
  kagi = '[';
  gika = ']';
  comma = ',';
  peri = '.';
  assign = ':=';
  colon = ':';
  semi = ';';
  eql = '=';
  noteq = '!=';
  bang = '!';
  le = '<=';
  lt = '<';
  ge = '>=';
  gt = '>';
  cand = '&&';
  cor = '||';
  iconst = sign? digit+;
  ident = alpha (alpha|digit|uscore)*;
  string = quote (nostr|escape)* quote;
  blank = (' ' | '  ' | 13 | 10)+;
  comment = '/' '/' [all - 10]* 10;

Ignored Tokens
  blank, comment;

Productions
  prog  = {emp}
        | {clus} prog clus
        | {mclc} prog mclu
        ;
  annot = {emp}
        | {one} annot atidn optcm
        | {lis1} annot atidn lpar nlist rpar optcm
        | {lis2} annot atidn kagi nlist gika optcm
        ;
  nlist = {emp}
        | {one} nlist nent optcm
        ;
  nent  = {idn} ident
        | {int} iconst
        | {str} string
        ;
  optcm = {emp}
        | {one} comma
        ;
```

```
 optsm = {emp}
       | {one} semi
       ;
 mclu  = {clu} ident eql metacluster cpara annot mstats end
       | {prc} ident eql metaproc cpara annot mstats end
       ;
mstats = {emp}
       | {one} mstats mstat
       ;
 mstat = {call} mcall optsm
       | {for}  for ident colon mexp do mstats end optsm
       | {ifst} if mexp then mstats melif end optsm
       ;
 melif = {none}
       | {else} else mstats
       | {more} elsif mexp then mstats melif
       ;
 mexp  = {call} mcall
       | {type} tname
       | {proc} [left]:dollar tname [right]:dollar ident
       | {sconst} string
       ;
 mcall = {none} mexp bang ident kagi gika
       | {args} mexp bang ident kagi margs gika
       ;
 margs = {one} mexp
       | {two} margs comma mexp
       ;
 clus  = {clu} ident eql cluster cpara annot vdefs pdefs end
       | {itf} ident eql interface cpara annot vdefs pdefs end
       ;
 cpara = {emp}
       | {one} kagi plist gika
       ;
 vdefs = {emp}
       | {one} vdefs vdef
       ;
 vdef  = {init} var ident colon tname assign expr
       | {decl} var ident colon tname
       ;
 pdefs = {emp}
       | {one} pdefs pdef
       ;
 pdef  = {func} ident eql proc lpar parms rpar colon tname annot fbody end
       | {void} ident eql proc lpar parms rpar annot vbody end
       ;
 parms = {emp}
       | {list} plist
       ;
 plist = {one} parm
```

```
      | {many} plist comma parm
      ;
parm  = {one} ident colon tname
      ;
tname  = {one} ident
      | {para} ident kagi tlist gika
      ;
tlist = {one} tname
      | {many} tlist comma tname
      ;
fbody = {emp}
      | {one} fbody fstmt optsm
      ;
vbody = {emp}
      | {one} vbody vstmt optsm
      ;
fstmt = {ascal} ascal
      | {retn} return expr
      | {whst} while expr do fbody end
      | {ifst} if expr then fbody felif end
      ;
felif = {none}
      | {else} else fbody
      | {more} elsif expr then fbody felif
      ;
vstmt = {ascal} ascal
      | {retn} return
      | {whst} while expr do vbody end
      | {ifst} if expr then vbody velif end
      ;
velif = {none}
      | {else} else vbody
      | {more} elsif expr then vbody velif
      ;
ascal = {vdef} vdef
      | {assign} ident assign expr
      | {astore} pmmex kagi [idx]:expr gika assign expr
      | {pstore} pmmex peri ident assign expr
      | {call} pclex
      ;
expr  = {orexp} orexp
      ;
orexp = {andex} andex
      | {cor} orexp cor andex
      ;
andex = {relex} relex
      | {cand} andex cand relex
      ;
relex = {addex} addex
      | {eql} [left]:addex eql [right]:addex
```

```
      | {noteq} [left]:addex noteq [right]:addex
      | {lt} [left]:addex lt [right]:addex
      | {gt} [left]:addex gt [right]:addex
      | {le} [left]:addex le [right]:addex
      | {ge} [left]:addex ge [right]:addex
      ;
addex = {mulex} mulex
      | {add} addex plus mulex
      | {sub} addex minus mulex
      ;
mulex = {uexpr} uexpr
      | {mul} mulex mult uexpr
      | {div} mulex div uexpr
      | {rem} mulex rem uexpr
      ;
uexpr = {calex} calex
      | {memex} memex
      | {minus} minus calex
      | {plus} plus calex
      | {bang} bang calex
      ;
args  = {empty} lpar rpar
      | {list} lpar alist rpar
      ;
alist = {one} expr
      | {list} alist comma expr
      ;
calex = {type} [left]:dollar tname [right]:dollar ident args
      | {memex} memex bang ident args
      | {casc} calex bang ident args
      | {prop} calex peri ident
      | {array} calex kagi expr gika
      ;
memex = {ident} ident
      | {const} const
      | {prop} memex peri ident
      | {array} memex kagi expr gika
      | {paren} lpar expr rpar
      ;
pclex = {type} [left]:dollar tname [right]:dollar ident args
      | {call} pmmex bang ident args
      | {casc} pclex bang ident args
      ;
pmmex = {ident} ident
      | {const} const
      | {prop} pmmex peri ident
      | {array} pmmex kagi expr gika
      ;
const = {iconst} iconst
      | {sconst} string
```

```
        | {false} false
        | {true} true
        | {nil} nil
        ;
```

4 標準クラスタの実装方法

4.1 標準クラスタ実装の概観

本稿で解説している o3 処理系は試験実装のためのものであり、標準クラスタ群の記述 (プレリュード) とソースコードを連結して 1 つのファイルとして入力している。標準クラスタの実装は特殊アノテーションとしてソースコード中に C 言語コードの形で書かれている (図 9)。ここでは o3 処理系が生成する C 言語コードの基本的なデザインと特殊アノテーションについて説明した後、具体的なソースファイルを示す。

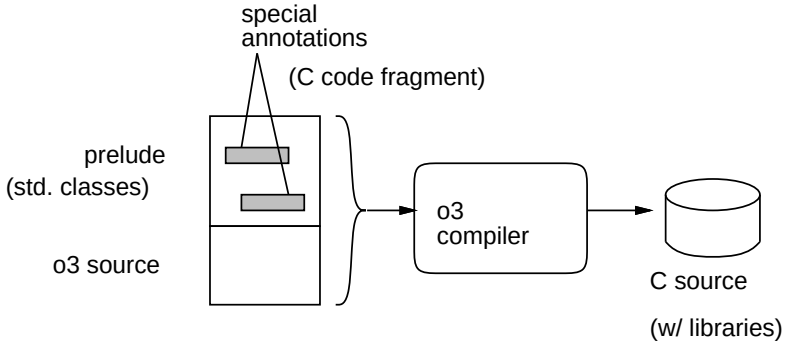

Figure 9: o3 ソースコードと特殊アノテーション

4.2 C 言語コードの概観

o3 処理系が出力する C 言語コードの冒頭部分を示す。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define nil 0
typedef void *ptr;
typedef ptr (*func)();
typedef struct vtbl_r {
  char *name;
  int bcnt, acnt;
  func *funcs;
} vtbl_r, *vtbl_p;
typedef struct obj_r {
  vtbl_r *ops;
} obj_r, *obj_p;
obj_p send0(obj_p self, char *name) {
  for(int i = 0; self->ops[i].name != 0; ++i) {
```

```
    if(strcmp(name, self->ops[i].name) == 0) {
      vtbl_p p = self->ops+i;
      int bc = p->bcnt, ac = p->acnt;
      for(int k = 0; k < bc; ++k)
        ((func)(p->funcs[k]))(self);
      ptr ret = ((func)(p->funcs[bc]))(self);
      for(int k = 0; k < ac; ++k)
        ((func)(p->funcs[bc+k+1]))(self);
      return ret;
    }
  }
  fprintf(stderr, "%s not sent.\n", name); exit(1);
}
```

ここで定義している識別子について説明する。

- `nil` — 0 だと出力が読みづらい箇所で `nil` と表記するため。
- `ptr` — ポインタ値の型。
- `func` — 関数ポインタの型。
- `vtbl_r` — 仮想関数テーブルの 1 エントリ。メソッド名、befre メソッド数、after メソッド数、および func の配列を挿す。
- `obj_r` — オブジェクトの表現。先頭に仮想関数テーブルの配列があり、その後ろは任意。

また、関数 `send`N はオブジェクトに対するメッセージ送信を行う下請け関数で、引数の個数により N が 0〜4 のものが生成される。オブジェクトが持つ仮想関数テーブルを探索して一致するメソッド名が見つかったらそれを呼び出す (現在は線形探索しているが、将来的には o3 コンパイラで固定番号を割り当て、探索を無くす予定でいる)。呼び出す際には、まず before メソッド (群)、次に本体、最後に after メソッド (群) の順で呼び、返値は本体メソッドのものを返す。メソッド名が見つからない場合はエラー終了するが、o3 コンパイラ側でメソッド名のチェックはしているので必ず見つかるはずである。

このほか、`send`N の類似の補助関数として `call`N、`callx`N も生成される。これらはメソッド名が解決済みの場合に使用されるので、直接関数テーブルエントリを受け取って呼び出す。`callx`N は `create` メソッドの場合に使われるが、現在のところ動作としては同じになっている。

個々のクラスタはすべて「識別子_i」というプレフィクスを持つ。ここで i はすべての型に割り当てる一意的な型識別番号 (TID) である (標準クラスタについては予め TID は処理系内で固定で割り付けられている)。パラメタつきクラスタの場合は、パラメタが異なればそれぞれ異なる TID となる。

4.3 特殊アノテーション

本節では標準クラスタの記述中で用いる特殊アノテーションについて説明する。

@clangnofwdcl

C 言語では前方参照ができないので、基本的に全クラスタについてまず型宣言を出力し、その後で各クラスの本体を出力する。これにより、各クラスの本体中で他クラスを参照して

も未定義にならずにすむ。しかし、標準クラスの一部は宣言そのものが循環参照になるため、型宣言を出力するとその一部に前方参照が含まれてしまう。このため、@clangnofwdcl のアノテーションがついたクラスタは型宣言を出力しない。なお、型パラメタつきのクラスタはもともと宣言を出力しない (C 言語コードには具象クラスタしか出現しない)。

@clangemit

各クラスタおよび各メソッドの宣言出力時に生成されるべき C コードを記述するアノテーション。ほかに@clangemit2 もあり、こちらは本体出力時 (宣言が終わった後) に生成されるものを記述する。

4.4 プレリュード部分のソース

本節では標準クラスタを記述するプレリュード部分のソースを示す。通常、この後ろに個別のコードが付加されている。

```
none = interface
  @clangnofwdcl
  print = proc(self:none) end
end

any = cluster
  @clangemit[
    "typedef struct string_4_r {"
    "  vtbl_r *ops;"
    "  int len;"
    "  char a[];"
    "} string_4_r, *string_4_p;"
    "extern vtbl_r *string_4_vtbl_copy;"
    "obj_p string_4_create(char *s);"]
  print = proc(self:any) "any"!print() end
  tostr = proc(self:any):string return "any" end
end

bool = cluster
  @clangnofwdcl
  @clangemit[
    "typedef struct bool_2_r {"
    "  vtbl_r *ops;"
    "} bool_2_r, *bool_2_p;"
    "extern bool_2_r bool_true;"
    "extern bool_2_r bool_false;"]
  @clangemit2[
    "bool_2_r bool_true = { bool_2_vtbl };"
    "bool_2_r bool_false = { bool_2_vtbl };"]
  equal = proc(self:bool, x:bool):bool
    @clangemit[
      "bool_2_p x1 = (bool_2_p)x;"
```

```
      "return (obj_p)((self==x1) ? &bool_true : &bool_false);"]
  end
  not = proc(self:bool):bool
    @clangemit[
      "return (obj_p)((self==&bool_true) ? &bool_false : &bool_true);"]
  end
  print = proc(self:bool)
    @clangemit[
      "printf((self==&bool_true)? \"true\\n\" : \"false\\n\");"]
  end
  tostr = proc(self:bool):string
    @clangemit[
      "return string_4_create((self==&bool_true)? \"true\":\"false\");"]
  end
end

int = cluster
  @clangnofwdcl
  @clangemit[
    "typedef struct int_3_r {"
    "  vtbl_r *ops;"
    "  int value;"
    "} int_3_r, *int_3_p;"
    "obj_p int_3_create(int v);"]
  @clangemit2[
    "obj_p int_3_create(int v) {"
    "  int_3_p x = (int_3_p)malloc(sizeof(int_3_r));"
    "  x->ops = int_3_vtbl; x->value = v; return (obj_p)x;"
    "}"]
  equal = proc(self:int, x:int):bool
    @clangemit[
      "int_3_p x1 = (int_3_p)x;"
      "return (obj_p)(self->value == x1->value ? &bool_true : &bool_false);"]
  end
  lt = proc(self:int, x:int):bool
    @clangemit[
      "int_3_p x1 = (int_3_p)x;"
      "return (obj_p)(self->value < x1->value ? &bool_true : &bool_false);"]
  end
  gt = proc(self:int, x:int):bool
    @clangemit[
      "int_3_p x1 = (int_3_p)x;"
      "return (obj_p)(self->value > x1->value ? &bool_true : &bool_false);"]
  end
  le = proc(self:int, x:int):bool
    @clangemit[
      "int_3_p x1 = (int_3_p)x;"
```

```
    "return (obj_p)(self->value <= x1->value ? &bool_true : &bool_false);"]
end
ge = proc(self:int, x:int):bool
  @clangemit[
    "int_3_p x1 = (int_3_p)x;"
    "return (obj_p)(self->value >= x1->value ? &bool_true : &bool_false);"]
end
minus = proc(self:int):int
  @clangemit[
    "return int_3_create(-(self->value));"]
end
plus = proc(self:int):int
  @clangemit[
    "return (obj_p)self;"]
end
add = proc(self:int, x:int):int
  @clangemit[
    "int_3_p x1 = (int_3_p)x;"
    "return int_3_create(self->value + x1->value);"]
end
sub = proc(self:int, x:int):int
  @clangemit[
    "int_3_p x1 = (int_3_p)x;"
    "return int_3_create(self->value - x1->value);"]
end
mul = proc(self:int, x:int):int
  @clangemit[
    "int_3_p x1 = (int_3_p)x;"
    "return int_3_create(self->value * x1->value);"]
end
div = proc(self:int, x:int):int
  @clangemit[
    "int_3_p x1 = (int_3_p)x;"
    "return int_3_create(self->value / x1->value);"]
end
mod = proc(self:int, x:int):int
  @clangemit[
    "int_3_p x1 = (int_3_p)x;"
    "return int_3_create(self->value %% x1->value);"]
end
print = proc(self:int)
  @clangemit[
    "printf(\"%%d\\n\", self->value);"]
end
tostr = proc(self:int):string
  @clangemit[
    "char buf[20]; sprintf(buf, \"%%d\", self->value);"
```

```
      "return string_4_create(buf);"]
  end
end

string = cluster
  @clangnofwdcl
  @clangemit
  @clangemit2[
    "vtbl_r *string_4_vtbl_copy = string_4_vtbl;"
    "obj_p string_4_create(char *s) {"
    "  int n = strlen(s);"
    "  string_4_p x = (string_4_p)malloc(sizeof(string_4_r)+n+1);"
    "  x->ops = string_4_vtbl; x->len = n; strcpy(x->a, s); return (obj_p)x;"
    "}"]
  equal = proc(self:string, x:string):bool
    @clangemit[
      "string_4_p x1 = (string_4_p)x;"
      "return (obj_p)(strcmp(self->a,x1->a)==0 ? &bool_true : &bool_false);"]
  end
  lt = proc(self:string, x:string):bool
    @clangemit[
      "string_4_p x1 = (string_4_p)x;"
      "return (obj_p)(strcmp(self->a,x1->a)<0 ? &bool_true : &bool_false);"]
  end
  gt = proc(self:string, x:string):bool
    @clangemit[
      "string_4_p x1 = (string_4_p)x;"
      "return (obj_p)(strcmp(self->a,x1->a)>0 ? &bool_true : &bool_false);"]
  end
  le = proc(self:string, x:string):bool
    @clangemit[
      "string_4_p x1 = (string_4_p)x;"
      "return (obj_p)(strcmp(self->a,x1->a)<=0 ? &bool_true : &bool_false);"]
  end
  ge = proc(self:string, x:string):bool
    @clangemit[
      "string_4_p x1 = (string_4_p)x;"
      "return (obj_p)(strcmp(self->a,x1->a)>=0 ? &bool_true : &bool_false);"]
  end
  add = proc(self:string, x:string):string
    @clangemit[
      "string_4_p x1 = (string_4_p)x;"
      "int n = self->len + x1->len;"
      "string_4_p y = (string_4_p)malloc(sizeof(string_4_r)+n+1);"
      "y->ops = string_4_vtbl_copy; y->len = n;"
      "stpcpy(y->a, self->a); strcat(y->a, x1->a); return (obj_p)y;"]
  end
```

```
  size = proc(self:string):int
    @clangemit["return int_3_create(self->len);"]
  end
  print = proc(self:string)
    @clangemit["printf(\"%%s\\n\", self->a);"]
  end
  tostr = proc(self:string):string
    @clangemit["return (obj_p)self;"]
  end
end

array = cluster[elt:any]
  @clangnoemit
  @clangemitgendcl[
    "typedef struct array_r {"
    "  vtbl_r *ops;"
    "  int limit, size;"
    "  ptr *body;"
    "} array_r, *array_p;"
    "void array_extend(array_p x) {"
    " ptr *b = x->body; int i = x->limit;"
    " x->limit *= 2;"
    " x->body = (ptr*)malloc(x->limit*sizeof(ptr));"
    " while(i >= 0) { x->body[i] = b[i]; --i; }"
    "}"]
  @clangemit[
    "extern vtbl_r array_%d_vtbl[];"
    "typedef struct array_%d_r {"
    "  vtbl_r *ops;"
    "  int limit, size;"
    "  ptr *body;"
    "} array_%d_r;"]
  create = proc():array[elt]
    @clangemit[
      " array_%d_p x = (array_%d_p)malloc(sizeof(array_%d_r));"
      " x->ops = array_%d_vtbl; x->limit = 8; x->size = 0;"
      " x->body = (ptr*)malloc(8*sizeof(ptr));"
      " return (obj_p)x;"]
  end
  size = proc(self:array[elt]):int
    @clangemit[
      " return int_3_create(self->size);"]
  end
  add = proc(self:array[elt], x:elt)
    @clangemit[
      " if(self->size >= self->limit) { array_extend((array_p)self); }"
      " self->body[self->size++] = x;"]
```

```
  end
  store = proc(self:array[elt], i:int, x:elt)
    @clangemit[
      " int_3_p i1 = (int_3_p)i;"
      " if(i1->value < 0 || i1->value > self->size) { return; }"
      " self->body[i1->value] = x;"]
  end
  fetch = proc(self:array[elt], i:int):elt
    @clangemit[
      " int_3_p i1 = (int_3_p)i;"
      " if(i1->value < 0 || i1->value > self->size) { return 0; }"
      " return self->body[i1->value];"]
  end
end
```

5 処理系の実装内部の概説

5.1 Compiler.java

コンパイラドライバは次の手順で動作する。

- ソースコードを SableCC によって生成されたパーサに解析させる。
- トラバーサに構文木を辿らせることにより、内部抽象構文木を生成させる。
- 抽象構文木上での意味検査を行う。
- 目的コード (C 言語のソースコード) を生成させる。
- (オプション) メインプログラム生成オプションが指定されていたら、指定されたメインメソッドを呼び出す C 言語の main 関数を生成。

5.2 Traverser.java

トラバーサは SableCC のコンベンションに従い、`DepthAdapter` クラスから継承し、各ルールの入口/出口に対応するフックメソッドをオーバライドすることで実現されている。各フックで実現しているのは基本的にソースコードに記された情報を保持する抽象構文木データ構造を構築することである。データ構造の各クラスには、受動的にデータを保持するだけのものもあるが、記号表を備えて意味検査の機能を提供するものも存在する。

5.3 ClangEmit.java

クラス `ClangEmit` は出力コード (C 言語ソース) 生成のための下請けクラスであり、文字列出力メソッドと標準ヘッダおよび下請け C 関数出力などのメソッドを持つ。

5.4 抽象構文木の受動的データ構造

5.4.1 Annots.java

クラス `Annots` はアノテーションの情報を保持するための受動的データ構造である。

5.4.2 MTreeList.java

クラス MTreeList は Mtree の並びを保持する受動的データ構造である。

5.4.3 StrList.java

クラス StrList は文字列の並びを保持するための受動的データ構造である。

5.4.4 TnameList.java

クラス TnameList は TypeName の並びを保持する受動的データ構造である。

5.4.5 TreeList.java

クラス TreeList は Tree の並びを保持する受動的データ構造である。

5.5 抽象構文木のオブジェクト

クラス Tree は抽象構文木の情報を保持するクラスである。内部には複数の補助クラスを持っている。

- Typecheck は木のノードを辿りながら型検査を行うビジタークラスである。
- Base は基底となるノードオブジェクトのクラスである。
- Vdef は変数定義ノードのオブジェクトである。
- Proc はメソッド定義ノードのオブジェクトであり、内部には宣言情報と文の並びを保持する。文の並びは Base の配列である。
- Para は型パラメタノードのオブジェクトである。
- Retn は return 文ノードのオブジェクトである。
- Whst は while 文ノードのオブジェクトである。
- Ifst は if 文ノードのオブジェクトである。
- Then は if 文内の then 節ノードのオブジェクトである。
- Else は if 文内の else 節ノードのオブジェクトである。
- Expr は式を表す抽象クラスである。
- Assign は代入式ノードのオブジェクトである。
- Call はメソッド呼び出し式ノードのオブジェクトである。
- Tcall は型指定のメソッド呼び出し式ノードのオブジェクトである。
- Not は論理否定ノードのオブジェクトである。
- Cand は論理 and ノードのオブジェクトである。
- Cor は論理 or ノードのオブジェクトである。
- Ident は識別子ノードのオブジェクトである。
- Iconst は整定数ノードのオブジェクトである。
- Bconst は論理定数ノードのオブジェクトである。
- Nil は定数 nil—を表すノードのオブジェクトである。

5.6 型環境のオブジェクト

5.6.1 TypeName.java

クラス TypeName は tid にマップできる状態になる前の (ソースコード上の) 型名を表現するのに用いる受動的データ構造である。

ここで o3 処理系内の型の表現について説明しておく。上述のように、TypeName はソースコード上の文字列表現と基本的同等である (構造を認識しているが)。

```
int
stack[T]   ←パラメタつきクラスタ
stack[int]
```

tid は型環境内のテーブルのインデックスとなる整数であり、標準クラスタについては処理系内で固定的に割り当てている。

```
3  ←  int
6  ←  stack[]
8  ←  stack[int]
```

パラメタつきクラスタでパラメタが指定されていないものも tid は持つが、具象型ではないので実体は生成できない。最後に正規化名はメッセージの表示などのために文字列の型名と tid をくっつけて生成した文字列表現である。

```
int_3
stack_6[]
stack_6[int_3]
```

5.6.2 TypeEnv.java

クラス TypeEnv は型番号 (tid) で識別される型本体 (Tbody.Base) の並びを管理する。内部にはこのため型本体のリストを持っている。また、それぞれの型について、親となる型の集合も保持している。

メソッド addBody は型本体を追加し、番号を割り当てる。型本体はパラメタが無い場合は自立した型、ある場合は型生成子となる。メ

ソッド idn2gen はパラメタ付き型の tid を割り当てる。既に同じ型生成子名・パラメタリストのものがあればそれを返す。無ければ型生成子の本体をコピーしてからそれにパラメタを追加して新たな型本体とする。そのほか内部データから検索するための複数のメソッド群もある。

メソッド check は型検査を行う。検査は自立した型を対象とし、種別ごとにその本体の check を呼び出す。

メソッド cemit はコード生成を行う。こちらは宣言部と本体部に分けてやはり種別ごとの cemit を呼び出す。

5.6.3 TypeEquate.java

クラス TypeEquate は型等値を扱うために TypeEnv の上にかぶせる薄い層である。環境によって別の等値が使える。大部分の仕事は内部に持つ TypeEnv を呼び出すだけだが、単純名の場合のみ、内部に持つ等値のマップを見てそこに等値定義があれば対応する置き換えを行う。

5.7 メソッド関係のオブジェクト

5.7.1 VarEnv.java

クラス VarEnv はメソッドやクラスタ内の変数定義群の環境を表現する。基本的には「< 識別子, 型名, 位置 >」の 3 つ組で、位置が負のものは外部のパラメタに対応する。

5.7.2 ProcType.java

クラス Proctype は 1 つのメソッド本体を表現するオブジェクトである。メソッド本体はコードを表す Tree.Proc と変数定義を表す Tree.Vdef の並びで表現される。また、メソッドには before/after メソッドが複数指定される場合がある。

メソッド colectVar 等はコード内から変数定義を集めて来るのに用いる。メソッド copy 等はメソッドのコピーを作る。メソッド checkAddCreate は create メソッド追加時のチェックをおこなう。cemit_body はコード生成用。

5.8 型本体 (Tbody.java)

型本体 (Tbody) は 1 つの型の情報を集約するオブジェクトである。実際には型本体の種別に応じて基底クラス Base から拡張した Mclu と Clus のいずれかになる。Base のインスタンス変数群は次の通り。

- paras — パラメタの抽象構文木の並び
- args — パラメタの型 ID の並び
- idn、cname、tname — 型名 (識別子のみ、正規化した文字列表現、TypeName オブジェクト
- annot — アノテーション
- te — 型定義内で使われる型等値オブジェクト
- pdic、pnum — 名前からメソッド、メソッド ID へのマップ

メタクラス、メタ手続きは入れ子クラス Mclu により表現される。インスタンス変数 proc が真ならメタ手続きとなる。メタクラスタの場合はコンパイル時にメタ操作が行われ、最後のコード出力時には通常のクラスタと同様にメソッドが出力される。

クラスタの場合はインスタンス変数として変数定義、メソッド定義、VarEnv を追加している。型検査時にメソッド辞書を構成してチェックし、最後のコード出力時にはメソッド本体を出力する。

5.9 メタ操作 (Mtree.java)

Mtree はメタ操作の抽象構文木のノードを定義している。Base が土台となるノードクラスであり、それを拡張する形で各種のノードが作られる。各種のノードは次の通り。

- Ifst — メタ if 文。
- Then — メタ if 文の then 部。
- Else — メタ if 文の else 部。
- For — メタ for 文。

- Msubr — メタ手続き。実際の各 API はこの後にメソッドとしてそれぞれ用意されている。
- Mex — メタ式。以下のノードはこのサブクラス。
- Mcall — メタ呼び出し。ここから上記手続を呼ぶ。
- Mtype — 型指定。
- Mproc — 手続き。
- Mstr — 文字列。
- Mval — 値。。以下が具体的な値のクラス。
- Vnull — null 値。
- Vstr — 文字列値。
- Vtype — 型名。
- Vproc — メソッド。
- Vtset — 型集合。
- Vpset — メソッド集合。

クラス TpEquate は外側の TypeEquate がそのままではメタ構文木内で使えないので皮をかぶせている。

6 処理系の使用方法

処理系はパッケージを取り寄せて展開すると、Java ソースコードと関連するファイル群、サンプルファイルが含まれている。処理系の本体は Makefile により jar ファイルとしてまとめられ、次のようにして動かせる (これを行うシェルスクリプト o3c も同梱)。

```
java -jar o3.jar ソースファイル
```

生成コード (C 言語ソース) は code.c という名称で生成される。

ただし、この方法で作った C 言語出力には main が含まれていないので、そのままでは実行できない。main を生成するには次のオプションを使用する。

```
java -jar o3.jar -m'$型名$メソッド名'  ソースファイル
```

これにより、指定した型の指定したメソッドを呼び出すだけの main が併せて生成されるようになる。

7 まとめ

本稿では o3 処理系の 2016.2.16 現在の現況について解説した。今後さらに言語機能の検討を進め、拡張していく予定である。

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。

References

[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, Concepts: 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.