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 composition mechanisms defeats design orthogonally of a language, and raises (miss-) selection problems. In this paper, we propose a programming language with a unified composition mechanism. In this mechanism, methods are individually copied and combined according to dedicated domain-specific language description executed at compile 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, composition mechanism, pluggable methods, strong types

1. Introduction

In today's procedural (imperative) programming languages, object-orientation (O-O) is one of major trends. In O-O, provision of abstract data types (ADTs) is an important functionality, in which (1) data (instance variables) and operations against them (methods) are grouped together to form a unit (an object), and (2) data are accessible only through accompanying methods, encapsulating objects' internals from client code.

In this paper, we use the term "abstraction unit" to represent description unit for an object instead of the common term "class," because we are excluding common functionalities seen in most class-based languages from the core of our language.

Programming languages' ADT functionality provides following merits in software development:

- Decrease dependencies among inside and outside of an abstraction unit, so they can be developed in parallel.
- Well-defined group of functionalities can be packaged as an abstraction unit and incorporated into the library, which enhances code reuse and development efficiency.

To enjoy those merits, an abstraction unit, once complete, should be incorporated "as is" into the client environment; any internal modifications should not be needed. Otherwise, internals of the abstraction unit and client environment will become interdependent and the merit noted above will vanish in the air.

In practice, the needs of the clients are much varied; there are many cases in which some customization to the existing abstraction units are necessary. In those occasions, instead of direct modification of the unit's internals, attaching some "correction" from outside will be more desirable, because we can refrain from breaking existing units and keep the merits of code reuse.

Such "correction" might be considered as a kind of unit, although it might not provide complete functionalities by its own. Then, customization can be regarded as a process of composing multiple (abstraction-) units.

Well-known composition mechanisms seen in today's O-O languages include inheritance, parameterization and AOP (Aspect-Oriented Programming). Historically, inheritance is the oldest, and parameterization and AOP arrived later to compensate weak points of the classic (inheritance-only) O-O languages.

However, having multiple composition mechanisms in single language will make the language complex, and poses problems of which mechanism to use in which case.

Therefore, in this paper we investigate unified framework of inheritance, parameterization and AOP for statically typed programming languages. As compile-time (static) composition is pursued, we exclude dynamic condition seen in some AOP languages (such as "cflowbelow" in AspectJ) from consideration.

Contents of this paper is as follows: In section 2, we focus on major composition mechanisms, namely inheritance, parameterization and AOP, and analyze their core functionalities. In section 3, we discuss paths toward unification of those functionalities, and describe our proposal. In section

[1] Graduate School of Systems Management, University of Tsukuba, Tokyo
[a)] kuno@gssm.otsuka.tsukuba.ac.jp

4, experimental programming language named "o3," with unified composition mechanism is presented. In section 5, we describe related works and compare our proposal against them, and finally in section 6, discussion and conclusion is presented.

2. Major Composition Mechanisms

2.1 Inheritance

In this section, we use the term "class" for abstraction unit, in order to make description simple. Inheritance is a functionality in which new class (subclass, child class) is defined upon existing class (superclass, parent class), with description of differences (extensions). Inheritance is the oldest of class composition mechanism; Simula, Smalltalk, C++, Java and many other O-O languages include inheritance.

Although details of inheritance mechanism differs among languages, common functionalities are as follows:

- Subclasses can add instance variables to the set defined in the superclass.
- Subclasses can add new method to the set defined in the superclass.
- Subclasses can replace implementation of methods defined in the superclass (or append / prepend additional code to existing method bodies in some languages).

Those functionalities allow definition of new classes by describing minimal difference against their superclasses; such style is called "differential programming."

On the other hand, differential programming is criticized for increasing interdependencies among class and thus maintenance difficulty. A solution for such criticism is preparing more general (abstract) differences as another class (called "mixins" or "traits"), and composing those classes with existing (parent) classes by means of multiple inheritance (inheriting from two or more parent classes).

In most of statically typed O-O languages, a class without parameter corresponds to a type. In such languages, a type corresponding to a child class (say C) becomes the subtype of its parents' type (say P), meaning that objects of type C can be used in place of type P object (including assignment to a variable). In this paper, we use notation $C \leq P$ to indicate subtype=supertype relations.

Subtyping allows a variable of type P to hold any of its subtype instances, and method invocation dispatches to whichever method code associated with current object held in the variable. This mechanism is called "dynamic dispatch" or "polymorphism" and is the source of large flexibilities seen in O-O languages.

On the other hand, there are criticism for such "binding" of implementation description (differential programming) and type compatibilities (subtype relations), and some languages try to separate these two aspects. Interface in Java languages is a representative one.

2.2 Parameterization

The term "parameter" is sometime used as "arguments" to individual operations (methods or procedures). However in this paper, the term "parameterization" stands for language functionality in which abstraction units can have (compile-time) parameters. Languages such as Java or Scala limits each parameter to a type, while other languages (C++, CLU[11]) also allows built-in type value (such as an integer) as a parameter.

In statically typed languages without parameterization, programmers are occasionally forced to write duplicate abstraction units (or stand-alone procedures) identical except for their operations' argument / return value types. It is natural desire to abstract out those difference as parameters and use a single definition for them.

Stated from a different viewpoint, the goal of parameterization is to abstract out type names that need not be bound to specific (concrete) types within an abstraction unit as parameters, so that more abstract and reusable description for an abstraction unit can be obtained.

A typical use of parameterization is for container abstraction units such as arrays. In an array type, content-type object is only stored within it and later extracted without any operation invocation, thus there are no constraints cast upon its parameter type.

However, some abstraction units might have constraints over their parameters. For example, in an ordered list, content-type objects should be comparable each other. Expression of those constraints has various form, depending on the languages:

- Operations' signatures are explicitly declared for each parameter — CLU.
- Interface or class hierarchy position are specified for each parameter — Java, Scala.
- Simply compile the whole unit after embedding parameter values, and OK when no error is encountered — C++.
- Module-like facility that specify constraints for parameters — proposed C++ concepts.[7]

In some languages, actual type for parameters are used to switch between distinct implementations for the abstraction unit (such as template specialization for C++). In languages which allow built-in type (e.g. integer) values as parameters, complex code construction through recursive parameter expansions are possible. C++ template metaprogramming is an representative one.[5]

Abstraction units with parameter defines a type when all of their parameters are specified. Therefore, an abstraction unit with parameter(s) defines a type generator. Rules of type inclusion relation (covariance / contravariance / nonvariance) among such parameterized types differ among languages. For example in Scala[12], type parameter definitions are annotated with variance specifications, and variance of the resulting types are deduced from them (examining usage of parameter types within their bodies).

Although Scala has type parameters (as noted above), inheritance can be used to realize same effect. Specifically, a class can include abstract type definition, and the type can be overridden with concrete type in its subclass, as in the following:

```
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
}
```

However, having multiple ways to do one thing in a language is against the principle of orthogonally; presence of such freedom is a matter of controversy.

2.3 AOP

AOP (Aspect Oriented Programming) means ways to provide functionalities (or "concerns") which are "crosscutting" to the structure provided by O-O class hierarchy. The term "aspect" is used to denote such crosscutting functionality defined separately from traditional classes.

The tasks such as thread synchronization, log recording, or redrawing of the screen are often cited as representatives of crosscutting concerns (difficult to fit with O-O hierarchy) and suitable to be implemented as aspects.

Major AOP languages or language mechanism include: AspectJ[8], SOP (Subject Oriented Programming[13]), Composition Filter[1], and Demeter/Adaptive Programming[10].

Functionalities provided by them can be summarized as: (1) specifying places on program execution paths (position on the code and time range in concern), (2) actions need be executed on those points, and optionally (3) additional variables and methods necessary to implement the actions. Therefore, an aspect consists of descriptions on (1) through (3) associated with a crosscutting concern.

Code positions noted above are often specified as entry / exit point of some method (both on caller / callee site), and method names (sometime specified through pattern) are frequently used to indicate which method is of concern. Additionally, some AOP systems provide dynamic (execution-time) conditions as when to invoke actions. As described previously, we exclude such dynamic conditions in this paper.

3. Unified Composition Mechanism

3.1 Preparation Toward Unified Mechanism

In this section, we discuss the policy toward our unified composition mechanism. As a prerequisite, we note that our mechanism is supposed to replace existing composition mechanisms (such as inheritance), and we would like to cover functionalities of existing composition mechanisms as much as possible. Therefore, we start from listing up what operations are performed to abstraction units with existing composition mechanism.

In case of inheritance, we reformulate the situation as composing parent unit and child unit to define new (inherited) unit. Within this framework, what inheritance does can be summarized as follows:

- Instance variables set of new unit is union of parent unit's set and child unit's set.
- Methods set of new unit is union of parent unit's set and child unit's set.
- In case of methods whose name appear both in parent's set and child's set, the method in child's set overrides one in parent's set.
- The type associated with the new unit is a subtype of the type associated with the parent unit.

In case of type parameterization, its functionalities are summarized as follows:

- Some of the names (representing types or values) which are referenced within the unit's body are declared as parameters, and specifying concrete types or values to them (instantiation) result in working unit with associated type.
- Interface (set of method signature) of a unit might depend on its type parameter in several ways, e.g. simply substituting parameter name with concrete types, or including set of method signatures from parameter types.
- Position of the resulting type associated with instantiated unit can be independent of its type parameters, or might depend on some of its type parameters in covariant / contravariant ways.

Finally, AOP functionalities are summarized as follows:

- Aspect-like unit includes description on where to and how to modify the target (modified) unit.
- In case of "where," method entry and method exit (both at caller and callee site) are representative.
- In case of "how to," specifying actions (groups of code) in the form of another method is the usual way.
- In some case, target (modified) unit is supplied with additional instance variables or method definitions (intertype declarations).

3.2 Basic Idea for Unification

From the above discussions, we saw that each of the existing composition mechanisms has both (1) operation on its associated type and (2) operation on its body (implementation). Our proposal in this paper is to incorporate DSL (domain specific languages) which describe above operations (1) and (2). The DSL description runs on behalf of the compiler and resulting (generated) units are processed by the compiler.

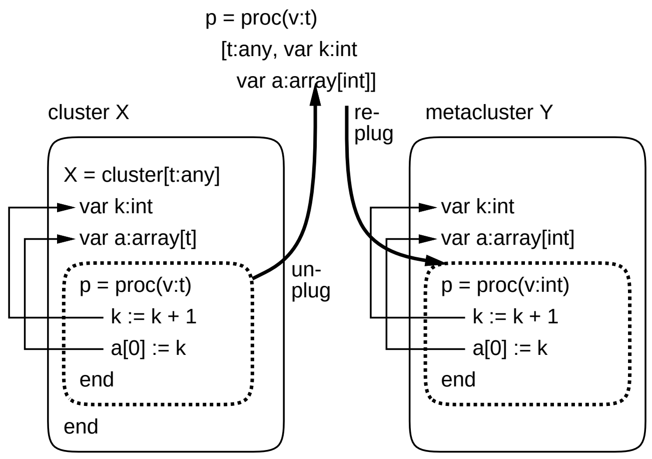

Fig. 1 Unplugging and plugging of a method

Our language will have traditional units, some include method signature only (as in Java interfaces) and some include method with associated body (as in Java classes); they are called base-level units. Other (meta-level) units includes DSL descriptions; DSL operations accept base-level language constructs (units, types, method signatures, method bodies) as passive data and builds new units based on them.

In the following, we describe our idea on both (1) and (2), respectively.

Operations Over Types

In this paper, based on the principles of abstract data types, we define a type as "a set of method signatures," where a method signature constitutes of method name, a list of types corresponding to its parameters, and its return value type (if one exist). Further, we assume that types has a supertype-subtype relation among them (thus form a type hierarchy). As the result, a type has the following operations:

(a) Define set of signatures corresponding to that type.

(b) Specify position of that type in the type hierarchy.

As for (a), we assume that a type has no signature associated when it is declared first, and there are DSL operations which selectively add existing signatures (obtained from other types), with modifications if necessary.

As for (b), we provide DSL operation which add new supertype-subtype relations to current type set. When those relations are arbitrary added, incompatible supertype-subtype pair might be formed (e.g. a subtype do not have operation signatures included in its supertype). Such situations are checked after DSL execution has finished and treated as compile-time errors (as the DSL is executed during compilation).

Operations Over Implementations

An abstraction unit contains a set of instance variable definitions and a set of method definition, and can create an instance (object). When a method associated with its originating abstraction unit is invoked, the body of the method is executed (that is, statements in the body is executed with necessary expression evaluation); instance variables of the object are accessed during the execution process.

```
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

Fig. 2 Summarized syntax of o3 language
(There are ; at the end of every statement, which are optional.)

In majority of existing O-O and AOP languages, modifications of existing code are performed through swapping by or appending / prepending of new code in the form of a method as a whole. Therefore, we decided to follow the same course and not to modify code inside a method; DSL operations act upon implementation in the following way (Figure 1):

(c) Extract a method as a whole from existing abstraction unit, and insert into the target (new) abstraction unit. There are choices of either replacing the existing method, or appending / prepending new method body to the existing method.

From the above description, it follows that each method belonging to an abstraction unit can be "unplugged" from the original context, and "replugged" to the new context. Conceptually, at the point of unplugging, references to the surrounding (instance-) variables and (parameter-) types are automatically converted to (variable- and type-) parameters, and those parameters are rebound when the body is replugged.

Therefore, parameter mechanism is built into our proposal language as one of the base functionalities, and inheritance or aspects are implemented with those functionalities. Such choice seems natural, because many abstract computational models (such as lambda calculus) include name substitution as primitive operation.

4. o3: An Experimental O-O Language

4.1 Design Guidelines And Program Structure

We have designed and developed a "concept-of-proof" O-O language named o3, in order to evaluate feasibility and effectiveness of the proposal described in the previous sections. Design guidelines of the language is as follows:

- Portions not directly related to the proposal should be similar to other "common" O-O languages.
- Portions not directly related to the proposal should be simple, as much as possible.
- Portions related to the proposal should clearly be separated from other part of the language.

A simplified syntax of o3 is shown in Figure 2. A program consists of one or more modules; a module is one of four kinds: interface, cluster (corresponds to class in other languages), metaprocedure, and metacluster.

An interface defines a type (set of method signatures), and a cluster defines a type and its implementation (set of instance variables along with set of method definitions). We use the term "cluster" in place of "class" because our cluster do not provide an inheritance facility; inheritance and other composition functionalities are provided through our proposal composition mechanism described below.

As noted earlier, we state type parameter as basic mechanism in our language. Therefore, both an interface and a cluster may have type parameter(s). A metaprocedure should have one or more type parameter(s) to act upon. As for metaclusters, meaning of presence / absence of type parameters is same as for clusters; a metacluster with type parameters define multiple types and their corresponding implementations according to the parameters.

Both metaprocedures and metaclusters contain type / cluster construction DSL (simply "DSL" for short); their syntax and functionalities are explained below. Note that execution of DSL occurs at compile-time.

4.2 Baselevel part of o3 language

In this section, we describe baselevel part (O-O without inheritance) of o3 language. In o3, method invocations are denoted with the form "*obj*`!`*method*`(···)`" or "`$`*type*`$`*method*`(···)`." The former corresponds to ordinary invocation with dynamic dispatching, and the latter to "static" invocation directly specifying typenames, which are used to create instances (in o3, method named "`create`" is handled specially and used to create new instances).

We list builtin interface / cluster in Figure 3. `any` is an interface, and is used as the supertype for all types in o3. `bool`, `int` and `string` are Boolean, integer and string values respectively; they are designated as builtin because they have literal forms, and `bool` is exclusively used for if / while conditions. `array` defines array types; it is designated as builtin for providing basic container object. `array` has an single type parameter to specify values stored the array.

```
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
```

Fig. 3 Interfaces for o3's builtin clusters

```
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
```

Fig. 4 A sample program in o3 (stack ADT)

Type hierarchy of o3 is shown in the Figure 5. Every type T is a (direct or indirect) subtype of `any` ($T \leq$ `any`). This policy is chosen because we need to define type parameters that accept arbitrary types. All other supertype-subtype

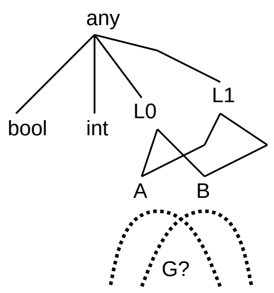

Fig. 5 Type hierarchy of o3

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*... `]`

Fig. 6 Summarized syntax of o3's metalevel DSL

Table 1 Metalevel objects and their DSL API (summary)

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

relations are explicitly defined through DSL operations.

As multiple supertypes can be designated for a type (through DSL operations), the relation $\leq$ forms a semiorder, and `any` becomes the maximum element. Therefore, for any types A and B, there always exist common upper bound type and their least elements (which may or may not be unique). On the other hand, for a pair of types A and B, their common lower bound may or may not exist; that depends on the cases.

In Figure 4, we show a simple o3 program. The code defines a stack ADT, then `main` create a stack object, pushes several values on it, popes some and prints. As explained above, `create` is handled specially in o3 — variable named `self` is automatically defined and holds newly created instance before execution of the method body.

4.3 Metalevel part of o3 language

As described earlier, metaprocedures and metaclusters include DSL descriptions, in the same syntax. Their difference is that metaclusters construct cluster definition, while metaprocedures are called from metaclusters with associated parameter(s) to execute series of DSL operations. Therefore, the objective of metaprocedures is to factor out common DSL operations with meaningful names, providing measure for structuring and abstraction.

As shown in figure 6, DSL code consists of metastatements. A simple metastatement has similar syntax as o3 baselevel method invocation, whose API is summarized in Table 1.

When a method is unplugged from existing cluster, instance variables and type parameters referred by the method body is automatically converted to variable / type parameters associated with the body (currently o3 does not have syntax to specify variable parameters directly). Thereafter, when the method is replugged to the other cluster being constructed, variable parameters are rebound to the instance variables of that cluster (new instance variables are automatically added if no such instance variable exist). Therefore, if one tries to replug multiple method with conflicting instance variables, an error is signaled. [*1]

Major metaobjects are types (set of method signatures with associated supertype set), clusters (same as types, plus method bodies) and methods (signature with optional body). There also is "method set" objects to handle groups of methods at once.

Additionally, string object (to specify method names and patterns) and Boolean object (to specify conditions for if metastatements) is also provided. Types of those metaobjects are dynamically checked at DSL execution stage, which is a part of the compilation stage.

There also are two compound metastatements, namely if and for. If is used to conditionally execute part of the operations. For metastatement is used to iterate over elements of a set.

4.4 Example: inheritance and logging aspect

In this section, we present an example with inheritance and logging aspect defined as metaprocedures (figure 7). The cluster `accum` defines an object that accumulate integer values. The cluster defines methods `create` (for creating an object), `inc` (for incrementing value) and `get` (for reading current value). Then, we want to define extended cluster which has additional method `reset`, which clears the value inside. In preparation, we defined a cluster named `exaccumimple` which includes implementation for the extension.

Actual inheritance operation is performed with a metaprocedure named `extends`, which receives three type parameters `target`, `parent` and `child` and copies methods defined in `parent` and `child` to `target`. In the metaprocedure body, `parent` is added to the set of `target`'s supertypes first. Then, set of methods defined in `parent` is copied to `target`, and then set of methods defined in `child` is copied likewise. As no selection or modification is required here, coping is done all at once (as set operations). Alternatively, for statement could be used to enumerate

[*1] Alternatively, we could provide renaming facility or simply treating those variables as distinct ones; such design choices are for future investigations.

```
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
```

Fig. 7 A sample with DSL code (inheritance and logging)

each method one by one (with selection or modification when necessary).

Next, we would like to count occurrence of modification operation (`add` and `reset`) invocations. This time, recording action is defined in another cluster `countimpl`. With its help, metaprocedure `addcounter` is used to modify the target cluster. First, the metaprocedure appends the body of `$countimpl$create` (which initialize count value) to the method `create`. Then, the metaprocedure copies signature and implementation of `getcount` (which obtains the count value). Finally, the metaprocedure enumerates all methods of the target cluster one by one, and for modification operations (distinguished by the method names in this example), the body of `countup` method is appended.

Actual extension object is defined by the metacluster `exaccum`; the metaprocedures `extends` and `addcount` are invoked from within its body. Within a metacluster, identifier "`selftype`" represents the type and cluster being defined by that metacluster.

4.5 Constraints applied at DSL execution

As shown above, our DSL allows flexible construction / modification of subtype relations, method signatures con-

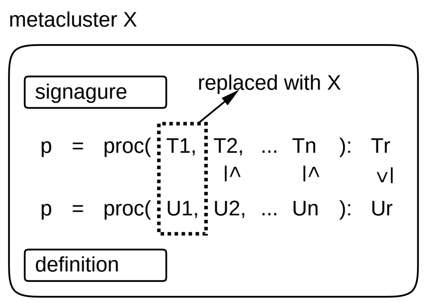
metacluster X
signagure
replaced with X
p = proc(T1, T2, ... Tn): Tr
|^ |^ v|
p = proc(U1, U2, ... Un): Ur
definition

Fig. 8 Signatures and compatibility of methods

tained in a type and method definitions associated with a cluster. However, as noted before, when there remains any inconsistency among resulting metaobjects, DSL runtime error (compile-time error for o3 compiler) is signaled. Actual processing and conditions need be satisfied are as follows:

- There should be no cycles in supertype-subtype graph.
- When a signature is added to a cluster, when the type of its first parameter is a supertype of the cluster's type, the type of first parameter is substituted by the cluster's type. [*2]
- When multiple method signatures with same method name are added to a cluster, the number of arguments and return value should match among them. Also, in the cases where substitution described above does not apply, the type of each argument becomes the minimal element of common upper bound of types for the corresponding argument of the signatures, and the type of return value becomes the maximal element of common lower bound of types for the return value of the signatures. When such minimal / maximal element is not uniquely determined, an error is signaled.
- When multiple method implementations with same method name are added to a cluster, the one added last survives (overwriting).
- When a method implementation is added to a cluster, method signatures with same name should already be associated with the cluster, and the implementation should be compatible with them. "Compatible" in this case means that each of the implementation、s argument type should be a supertype of corresponding signature's type, and the implementation's return type (if any) should be a subtype of signature、s return type (Figure 8). [*3]
- When before or after methods are added to a primary method, the number of arguments for methods being added should be less or equal to the number of arguments for the primary method, and existing argument's types should satisfy conditions described above. (Non-)

[*2] The reason for such substitution is that the first parameter plays the role of receiver, over which dynamic dispatch is done. The choice of not applying such substitution is also provided through another metaobject API calls.

[*3] We are planning to supplement DSL API with functionality to insert conversion code when those compatibility constraints.

Table 2 The size of o3 implementation

	#. of lines
SableCC grammer	255
Java code	2400

existence of return values are arbitrary (return value is ignored). [*4]

5. o3 Implementation

We use SableCC[6] compiler-compiler for lexical / syntax analysis, along with its syntax tree construction / traversal facilities. Other portions of the compiler are written in Java; we show the code sizes in Table 2.

In the semantic analysis phase, information from interfaces and clusters are gathered in the type table, along with their syntax tree for method bodies. As for metaprocedures and metaclusters, DSL syntax trees are stored in the same type table, and then interpreted execution of metacluster DSLs are performed. During the DSL execution for a metacluster, empty type data structure is first created and then modified according to DSL description, and resulting structure is type- and semantic-checked at the final stage. Finally, code generation is performed for both baselevel clusters and metaclusters (Figure 9). Current compiler is an experimental one and do not support separate compilation.

The compiler generates code in plain C language, and after the code is compiled by a C compiler, the code can be executed. Every object has a method table pointer as the first component, and instance variables part (dependent on the object's type) follows. A method table stores pointer to method vector, and a method vector points to bundle of code pointers (C language function pointer). In case of a method with primary portion only, its method vector contains a single code pointer, and when before / after code are added, corresponding method vector stores list of code pointer in execution order. The number of before / after methods (code pointers) are stored in the corresponding method table entry.

Implementations for the builtin clusters are described as special (system-only) annotation, and C language code for them are generated and prepended to the C language output prior to the code generation for actual program. For storage management, we just use conservative GC[3].

6. Related Works

Although there are many research on inheritance, type parameters (generics) and AOP, unification of these language mechanisms is not much investigated.

As described before, Scala[12] allows inheritance to function as type parameterization through overriding abstract type member(s) of parent classes on their subclasses. However, Scala language design do have type parameterization by itself and does not intend to unify it with inheritance.

[*4] We are planning to supplement DSL API with functionality in which after methods can accept the return value form the primary (or previous after) method, process it, and return substitute value.

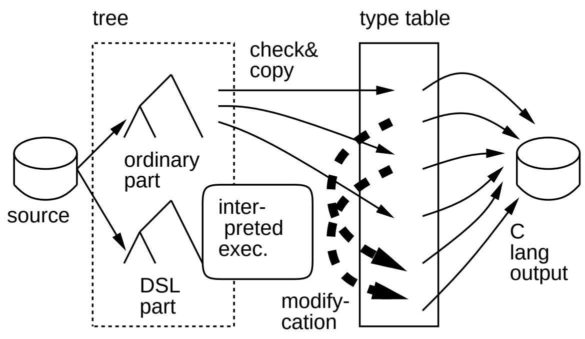

Fig. 9 The structure of o3 compiler

Additionally, inheritance itself is a much complex and (too-) powerful language mechanism; aim of our research is to decompose inheritance into more primitive functionalities.

Bergmans et. al.[2] are proposing a language which unify inheritance and AOP with execution-time (dynamic) property deduction. However, their proposed framework does not consider static typing, and incur much overhead on method dispatch due to dynamic computation; we are aiming for more static and efficient mechanism with compile-time type checking, which we believe is important for building robust systems.

Controlling method dispatch and inheritance operation can also be performed through metaobject protocols (MOP); [9] and [4] are representatives ones. However, MOP is based on inheritance as a basic mechanism, and controlling code are injected by subclassing existing meta-objects and overriding some of the meta-methods (such as meta-method invocation); they are not aiming at replacing inheritance with set of more primitive operations.

7. Discussion and Conclusion

In this paper, we have proposed a programming language with an unified composition mechanism, with which inheritance, type parameterization, AOP and similar mechanism can be constructed.

In our proposal, parameter substitution is built into the language core, and compositions are performed through extracting and combining signatures and method implementations from existing abstraction units. To make such operations possible, our language possess dedicated meta-level DSL (in addition to ordinary — base-level — language core). Our compiler executes DSL description at compile time (with an interpreter built into the compiler), through which composition operations are actually performed.

To asses practicality and problems of the above scheme, we have designed and implemented an experimental object-oriented programming language "o3" as a test bed. First experiences from this language is that such language can actually be built, and can be used a lot like ordinary (existing) O-O languages.

However, simple declaration such as "extends *superclassname*" in existing O-O languages have to be replaced with somewhat more lengthy code at the using sites, inheritance implementation codes set aside. Major complication is that implementation of additional parts (additional methods or overriding methods) have to be described separately from the metacluster which corresponds to “subclass” unit. If those two portions could be described as a single unit, the language will look much similar to existing O-O language (with respect to inheritance usage).

We could write AOP-like functionalities (excluding controls with dynamic properties) without much difficulty. However, injection specifications (“pointcuts” in AspectJ terms) are currently limited to pattern matching with respect to method names. More general and flexible design would be to add annotations to methods and use them for injection specification. When adding annotation at source-code level is undesirable, DSL API could be enhanced with additional functionalities that examine abstraction units and method signatures/implementations and attach appropriate annotations.

Current DSL API choices are minimal because we have proposed small proof-of-concept implementation. We are going to investigate more powerful API and their semantics so that various useful language mechanisms could be built using them, in type-safe manner.

This work was supported by JSPS KAKENHI Grant Numbers 25330076.

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.