Maintenance of Object-oriented Systems during Structural Evolution *

Paul L. Bergstein
Department of Computer Science and Engineering
Wright State University
Dayton, OH 45435

Keywords: Object-oriented Software Engineering, Structural Evolution, Program Transformations

We have previously developed a mathematical treatment of a calculus for class transformations that preserve or extend a set of objects. Methods for automating the maintenance of structural and behavioral consistency in systems based on evolving class structures have been provided for the object-preserving and object-extending transformations. This work extends the calculus of class transformations to include certain transformations that reflect not only the extension and reclassification of existing objects, but also structural changes (other than addition of attributes) in the original objects.

Language-preserving transformations are a special case of transformations that change the structure of existing objects. If an object schema is decorated with concrete syntax, it defines not only a class structure, but also a language for describing the objects. When two schemas define the same language but different classes, the language may be used to guide the transportation of functionality between domains. The language-preserving transformations defined here form the basis of a complete transformation system for a subset of class graphs powerful enough to express the regular languages.

1. Introduction

Class organizations (schemas) evolve over the life cycle of object-oriented systems for a variety of reasons. This issue has recently been a subject of increasing attention in the literature of both object-oriented languages and especially object-oriented database systems: [22, 19, 29, 8, 5, 10, 11, 12, 4, 21, 1, 3, 30, 34].

One of the most common forms of evolution involves the extension of an existing schema by addition of new classes of objects or the addition of attributes to the original objects. Sometimes class structures are reorganized even when the set of objects is unchanged. In this case the reorganization might represent an optimization of the system, or just a change in the users' perspective. At the other extreme, a class reorganization might reflect not only the extension and reclassification of existing objects, but also structural changes (other than addition of attributes) in the original objects.

While extension and reclassification of objects are mainly concerned with the organization of objects into classes, object restructuring is concerned primarily with the organization of attributes, or "parts", into objects. Here, a major concern is how to modify the code of an object-oriented program if the class definitions are changed so that the same data is organized into a different object structure. If the new objects hold the same data as the original objects, the class structures can be considered in some way analogous. The problem is to find a mapping of the code (methods) from the old class structure to the new one.

Some of the issues related to maintaining behavioral consistency during object restructuring have been previously addressed by Johnson and Opdyke [19] and others. In this paper, we consider a novel framework for dealing with object restructuring with a theoretical basis in formal languages. We view the original and restructured class hierarchies as structurally different grammars that define the same language.

The programming language model used throughout the remainder of this paper is formally defined in section 4. In the model, a program comprises a set of class definitions in the form of a class graph plus a set of method definitions. The data model, formally defined in appendix A, is an extension of the Demeter Kernel Model [24] which allows decoration of class graphs (schemas) with concrete syntax.

The extended data model uses a graphical notation to define both a set of objects and a language for representing the objects textually. In a class graph squares represent concrete classes and hexagons represent abstract classes. Inheritance (subclass) relationships are indicated by wide arrows. Thin arrows are used to specify the attributes and concrete syntax, collectively referred to as "parts", associated with a class.

*This research was partially supported by Ohio Board of Regents grant 663019.

The parts of a class (including inherited parts) are totally ordered. The textual representation of an object is obtained by concatenation of the concrete syntax as it is encountered during a depth first traversal of the object's parts.

An interesting class of transformations investigated in section 5 are those that change the structure of the objects, but preserve the language defined by the schema. Since a class graph defines both a class structure and a grammar, any change in the class structure is reflected in a corresponding change in the grammar. There is an interesting class of transformations that result in a new class structure, a potentially new set of objects, and a new grammar, but which leave the *language* defined by the grammar unchanged. I call such a transformation *language-preserving* and say that the old and new class graphs are *language-equivalent*.

Example 1. The class graphs in Figure 1 are language-equivalent even though they define different sets of objects. Grammars corresponding to the class graphs ϕ_1 and ϕ_2, respectively, are given below in EBNF form:

```
<Prefix> ::= <Number> | <Compound>
<Number> ::= Digit {Digit}
<Compound> ::= <AddExp> | <MulExp>
<AddExp> ::= '(' '+' <Prefix> <Prefix> ')'
<MulExp> ::= '(' '*' <Prefix> <Prefix> ')'
```

```
<Prefix> ::= <Number> | <Compound>
<Number> ::= Digit {Digit}
<Compound> ::= '(' <Op> <Prefix> <Prefix> ')'
<Op> ::= <AddOp> | <MulOp>
<AddOp> ::= '+'
<MulOp> ::= '*'
```

If the concrete syntax specified in a schema is meaningful and a transformation preserves the defined language, it is reasonable to hypothesize that the objects are intended to represent the same data in the transformed schema as in the original. If the inputs and outputs of a program are objects, then it is reasonable to expect that the code could be automatically updated after a language-preserving transformation so that any input in the language will produce output identical to the output from the original program[1]. In other words, we can expect to find a mapping of the methods from the old class structure to the new one which will preserve the behavior of the system.

These techniques may be useful in contexts other than evolution, e.g. during implementation when it is desirable to reuse some of the functionality of an old application in a new environment.

2. Motivating example

Consider the first class graph, ϕ_1, in Figure 1. Since the `Prefix` class is abstract, every `Prefix` object must be an instance of a concrete subclass: `Number`, `AddExp`, or `MulExp`. `Number` objects are represented textually by strings of digits. `AddExp` and `MulExp` objects are represented textually by strings comprising an opening parenthesis, a "+" or "*", the textual representations of their two subexpression arguments, and a closing parenthesis. They may be nested to any arbitrary depth.

In the second class graph, ϕ_2, the `Compound` class has been made concrete. We no longer have subclasses of `Compound` to distinguish between multiplication and addition. Instead, we use an attribute, `op`, which takes as its value an instance of an `AddOp` or `MulOp`. Notice, however, that the textual representations of `Compound` objects has not changed.

Suppose we start with ϕ_1 and implement a program to evaluate prefix expressions in an object-oriented language such as C++. The class definitions can be automatically generated from the class graph. More interestingly, code to parse the program's input and build the corresponding `Prefix` object can also be automatically generated. The application is completed by adding a few simple methods:

```
int Number::eval()
{ return value; }

int AddExp::eval()
{ return (arg1->eval() + arg2->eval()); }

int MulExp::eval()
{ return (arg1->eval() * arg2->eval()); }
```

Now, suppose we wish to change the class structure to that of the second class graph, ϕ_2. We start by rerunning the code generator to get a new set of class definitions and new code to parse and build `Prefix` objects.

When parsing a `Compound` in the original program, an instance of either `AddExp` or `MulExp` is created, depending on whether a "+" or "*" is found in the input stream. When a `Compound` is parsed in the new program, an instance of the concrete `Compound` class is created. This involves parsing the `op` part of the `Compound` to produce either an `AddOp` or `MulOp` depending on whether a "+" or "*" is found in the input stream. Wherever an object had an `AddExp` or `MulExp` as a part in the original program, the corresponding object will have a `Compound` with either an `AddOp` or `MulOp`, respectively, in the transformed program.

In order to maintain functional equivalence, it is necessary for a `Compound` object in the transformed program to pass along any message it receives to its `op` part with the extra argument `this`. Methods written for the original program are mapped from `AddExp` $\rightarrow$ `AddOp` and from `MulExp` $\rightarrow$ `MulOp`. The only modification to the methods is that they must access any parts defined for `Compound` objects indirectly[2] through the extra argument:

```
int Number::eval()
{ return value; }

int Compound::eval()
{ return op->eval(this); }
```

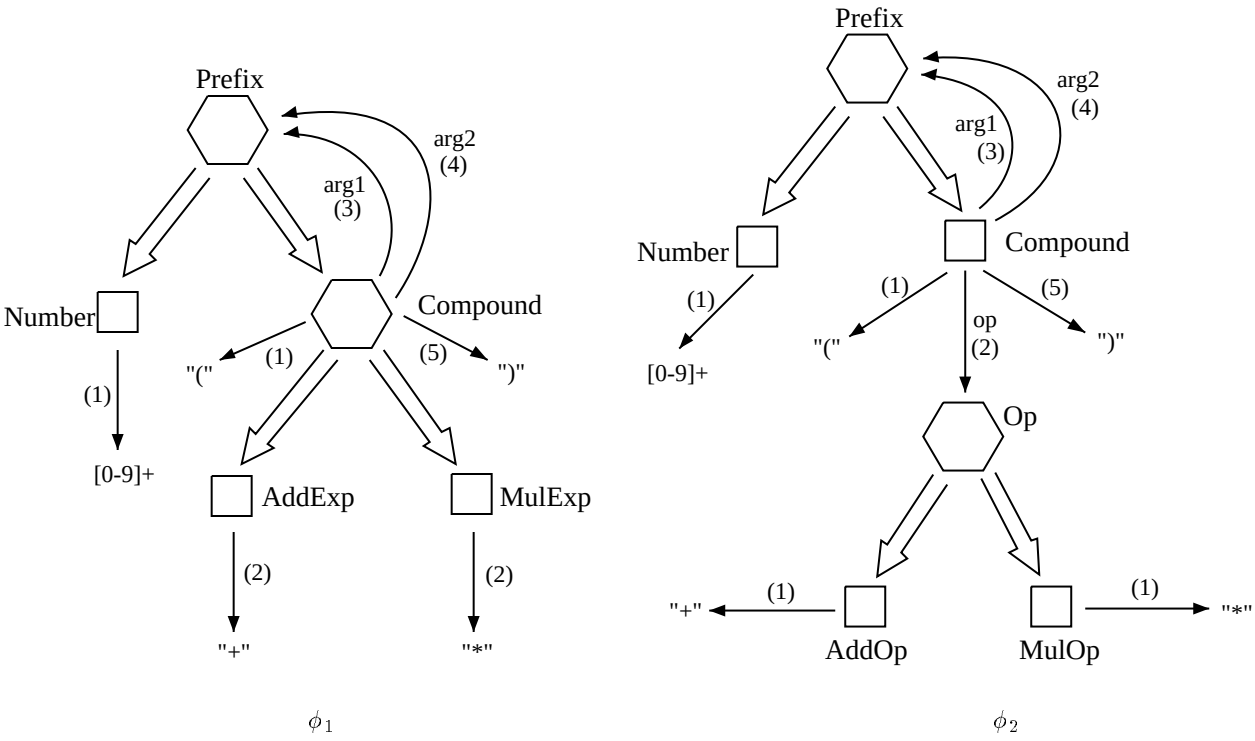


FIG. 1. Language-equivalent Class Graphs

```
int AddOp::eval(Compound* exp)
{ return (exp->get_arg1()->eval() +
          exp->get_arg2()->eval()); }

int MulOp::eval(Compound* exp)
{ return (exp->get_arg1()->eval() *
          exp->get_arg2()->eval()); }
```

The subclass to attribute transformation and its inverse are just two of many common class transformations that change the structure, but not the textual representation or information content, of objects. The remainder of this paper is dedicated to developing methods for *automatically* restoring the behavior of a system after such class transformations.

3. Research Approach

The approach taken is to break the problem down into three manageable sub-problems:

1. **Defining a set of primitive transformations.** A small set of primitive transformations is defined which can be composed sequentially to form many useful language-preserving transformations. By *useful*, we mean those transformations that would make sense from a software design point of view. The primitives defined in Section 5 allow transformations including:
   - Any transformation that preserves the original objects
   - Renaming of classes and attributes
   - Addition and deletion of useless symbols
   - Distribution of parts up or down the part-of hierarchy
   - Replacing subclasses with attributes or attributes with subclasses
2. **Providing algorithms for incrementally updating the code.** For each primitive transformation an algorithm must be found for updating the code. Then, given any sequence of primitive transformations, the code can be updated incrementally by performing updates for each of the primitives in sequence.
3. **Reducing an arbitrary language-preserving transformation to a sequence of primitives.** An algorithm to search for a sequence of primitive transformations between two arbitrary language-equivalent class graphs must be found. The search may be effectively guided by the concrete syntax of the language. When the search is successful, we may regard the resulting sequence of primitives as the definition of an analogy between the class graphs.

4. Language Model

This section describes a simple object-oriented programming language based on class graphs. The CG language will be used to illustrate the method transformations that are required to restore behavioral consistency after a language-preserving class graph transformation. Although the CG

language is very simple, the same principles can be used to modify programs written in "real" languages.

4.1. Overview

A CG (class graph) program consists of a set of class definitions in the form of a class graph plus a set of method definitions. There is one built-in class called `Number` for which the language provides the built-in methods `add`, `sub`, `mul`, `div`, `assign`, and `print`. The user may define still more methods for the `Number` class.

The class definitions must include a concrete class called `Main`, and the method definitions must provide a `main` method for the `Main` class. Program execution begins by parsing the input by recursive descent to construct an instance of the `Main` class (the main object), and invoking its `main` method. Note that programs written in the CG language are self documenting as to their legal inputs since they define their own input language.

In the CG language, there is no way to create or destroy objects once the initial parsing operation is complete. Thus, the set of objects is fixed during program execution and consists of a tree rooted at the main object.

Each object except the main object has an implied "container" attribute. An object's container is its parent in the object tree. In other words, the attribute links between objects are defined to be bidirectional. This feature of the CG language is included to simplify some of the code transformations discussed below. In a "real" language, the container attributes[3] would be implemented only when required by a code transformation.

In the CG language, as in languages such as Smalltalk, each object has direct access only to its own attributes. However, in CG these attributes include the container attribute. In other words, each object has access to its own parts and to the object of which it is a part.

4.2. Methods, Messages, and Expressions

Each method may take any number of objects as arguments and every method returns an object. Both the arguments and the return value are passed by reference. This must be the case, since passing by value would involve the construction of new objects during program execution.

method ::= class : name '(' formals ')' '{' explist '}'
formals ::= name { , name }
This construct is a method definition. When the method is invoked by sending the *name* message to an object of class *class*, the expression *explist* is evaluated after argument values are substituted for the formals and its value is returned.

explist ::= exp { ; exp }
An expression list is evaluated by evaluating each expression in the list from left to right. The value of the list is the value of the last expression.

exp ::= name
Here, name may be either the name of a part of the object for which the method being evaluated was invoked or the name of a formal parameter of the method. The value of the expression is the object instantiating the part or the object passed as the actual argument, respectively.

exp ::= self
The value of the expression *self* is the object for which the method being evaluated was invoked.

exp ::= container
The value of the expression *container* is the object which contains the object for which the method being evaluated was invoked. The expression *container* must not appear in any method attached to the Main class.

exp ::= exp ← name '(' [actuals] ')'
actuals ::= exp { , exp }
This construct denotes the sending of a message. When an object is sent a message, the object's method called *name* is invoked. If the object has no method with the proper name, a run time exception occurs and program execution is terminated. The evaluation order is: The *exp* on the left hand side of the message send operator, ←, is evaluated; each of the actual argument expressions is evaluated and their values are substituted for the corresponding formals in the method body; the method body is evaluated and the result is returned.

Note that the CG language supports delayed binding but not inheritance of methods. Inheritance is not an important issue in the study of code transformations since it can easily be eliminated from an object-oriented program just by copying methods from superclasses to the classes where they are inherited. The inheritance mechanism is merely a convenience for the programmer so that each method only needs to be written in one place.

4.3. Built-ins

All of the built-in methods for the `Number` class except `print` take a single argument which must be another `Number`. All of the methods return `self`. A side effect of the methods `add`, `sub`, `mul`, `div`, and `assign` is that the "value" of the `Number` object receiving the message is changed. That is, the state of the object is changed in such a way that subsequent messages to the object may have different results. The value is modified in the obvious way depending on whether the message is `add`, `sub`, `mul`, `div`, or `assign`. When a `Number` is created during the initial parsing, its value is initialized depending on the value denoted by the token parsed. When a `Number` receives the `print` message, a token denoting its value is output.

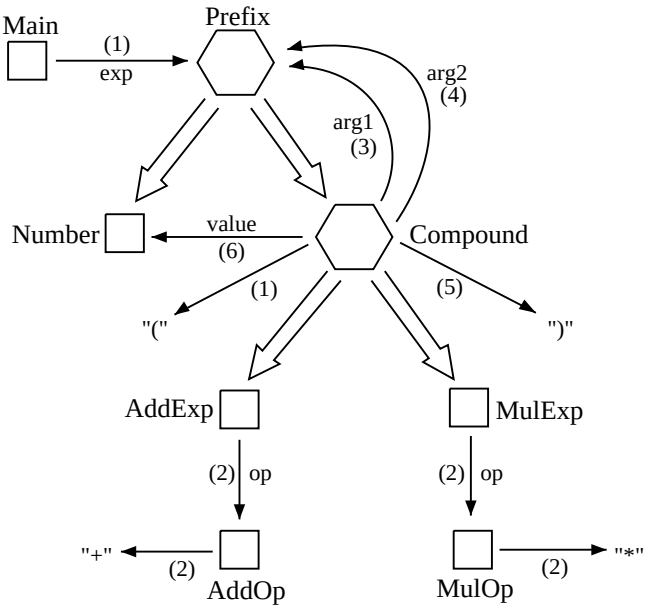
Example 2. A complete CG program to evaluate arithmetic prefix expressions is shown in Figure 2.

5. Language-preserving Class Transformations

5.1. Primitive Language-Preserving Transformations

The following primitives comprise the language-preserving class graph transformations, and are described informally below. Formal definitions can be found in Appendix B.

1. The object-preserving transformations
2. Renaming of vertices and edges
3. Nesting of parts
4. Unnesting of parts
5. Addition of lambda parts
6. Deletion of lambda part
7. Addition of lambda alternative
8. Deletion of lambda alternative
9. Insertion of singleton construction
10. Deletion of singleton construction
11. Attribute to subclass
12. Subclass to attribute




```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

AddExp : eval ()
{
   value <- assign(arg1 <- eval());
   value <- add(arg2 <- eval())
}

MulExp : eval ()
{
  value <- assign(arg1 <- eval())
        <- mul(arg2 <- eval())
}
```

FIG. 2. Program A

5.1.1. Object-preserving transformations The object-preserving transformations for class graphs are almost the same as the object-preserving transformations defined in [5] for graphs lacking concrete syntax and ordering of parts. There is an additional primitive that allows edges to be renumbered as long as the ordering is unchanged. *Abstraction of common parts* and *distribution of common parts* are extended to apply to syntax edges as well as attribute edges. The only additional complexity is that abstraction of common parts to a superclass is restricted so that the ordering of parts at each immediate subclass cannot be changed. If there is a set of classes that have more than one part in common, but the common parts are ordered differently in the individual classes, then it is not possible to abstract all of the common parts.

The object-preserving transformations for class graphs are as follows:

- **Renumbering of parts**. Any set of attribute and syntax edges in a class graph may be renumbered as long as the ordering of parts (including inherited parts) remains unchanged for each class.
- **Abstraction of common parts**. If all of the immediate subclasses of class C have the same part, that part can be moved up the inheritance hierarchy so that each of the subclasses will inherit the part from C, rather than duplicating the part in each subclass.
- **Distribution of common parts**. This is the inverse of abstraction of common parts.
- **Deletion of "useless" alternation**. A vertex representing an abstract class (formally, an *alternation vertex*) is "useless" if it has no incoming edges and no parts. Intuitively, an abstract class is useless if it is not a part of any concrete class, and it has no parts for any concrete class to inherit. If an abstract class is useless it may be deleted along with its outgoing inheritance edges.
- **Addition of "useless" alternation**. An abstract class can be added along with outgoing inheritance edges to any set of classes already in the class graph. This is the inverse of deletion of useless alternation.
- **Part replacement**. If two classes, C1 and C2, have the same set of instantiable (concrete) subclasses then any attribute edge incoming at C1 may be rerouted to C2, since the set of objects which may instantiate the attribute will not change. Note that the inverse of part replacement is just another instance of the transformation.

5.1.2. Renaming of vertices and edges Classes and attributes may be freely renamed.

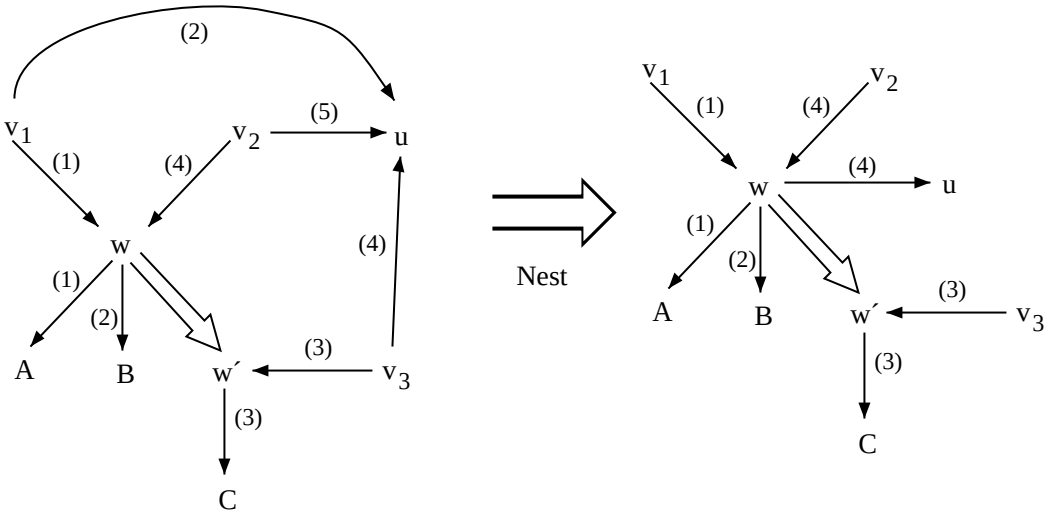


FIG. 3. Nesting of parts

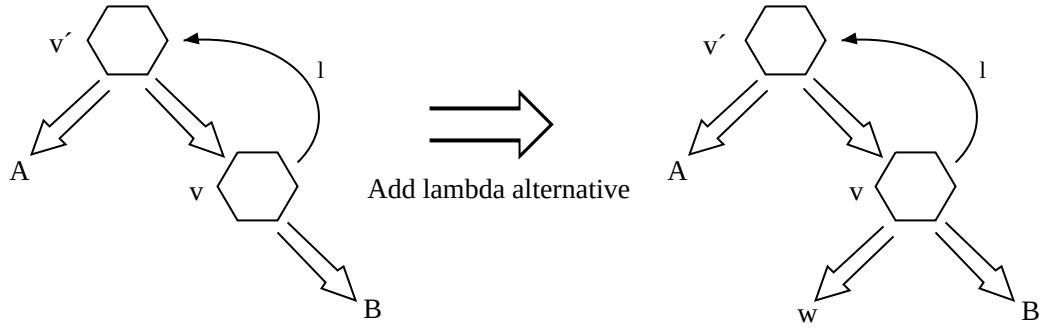


FIG. 4. Addition of lambda alternative

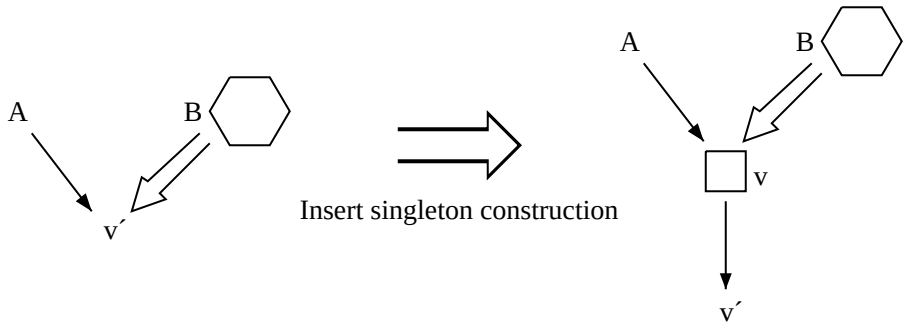


FIG. 5. Insertion of singleton construction

5.1.3. Nesting of parts

- If every class which has a class w as a part, has u as a part immediately after w, then we may remove the u part from all of those classes and instead make u the last part of class w. See, for example, Figure 3.
- If every class which has w as a part has u as a part immediately before w, then we may remove the u part from all of those classes and instead make u the first part of class w.

5.1.4. Unnesting of parts This is the inverse of *nesting of parts*.

5.1.5. Addition of lambda parts A part, p, can be added to any class if p is a class with no parts and no subclasses, or if p is the "empty string".

5.1.6. Deletion of lambda parts This is the inverse of *addition of lambda parts*.

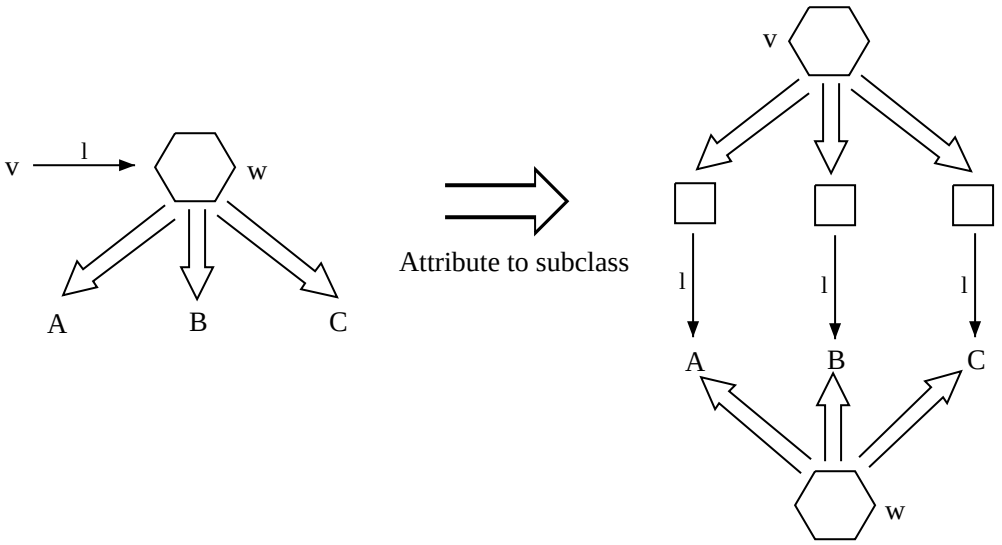


FIG. 6. Attribute to subclass

5.1.7. Addition of lambda alternative If an abstract class, v, has as its only immediate (not inherited) part an immediate superclass of v, v', then a concrete class, w, with no parts may be added to the subclasses of v. See, for example, Figure 4.

5.1.8. Deletion of lambda alternative This is the inverse of *addition of lambda alternative*.

5.1.9. Insertion of singleton construction A new concrete class, v, with a class, v', as its only part may be added to a class graph, and v may replace v' as a part in any other class. If v' is a subclass, inheritance edges may be rerouted from v' to v if the change does not result in the inheritance of any additional parts at v. See, for example, Figure 5.

5.1.10. Deletion of singleton construction This is the inverse of *insertion of singleton construction*.

5.1.11. Attribute to subclass If a class graph contains a concrete class, v, with an abstract class, w, as a part, then we may delete the part, w, from v and for each immediate subclass, w', of w we create a new concrete class, v' with w' as a part, and make v' a subclass of v. Class v becomes abstract. See, for example, Figure 6.

5.1.12. Subclass to attribute This is the inverse of *attribute to subclass*.

5.2. Justification for the primitive transformations

One justification for the selection of the chosen primitives is that they make it possible to express commonly occurring language-preserving transformations as a sequence of primitives. Examination of the literature and personal experience with the evolution of the Demeter system indicate that the primitive transformations defined in this section are powerful enough to express most, if not all, of the transformations that could be considered language-preserving. Rather than argue the subjective "practical usefulness" of the transformations, however, we demonstrate that the primitive language-preserving transformations defined here form the basis of a *complete* transformation system for a subset of class graphs powerful enough to express the regular languages.

5.2.1. Regular class graphs The *regular* class graphs are defined so that there is a bijection between them and the regular expressions. The alphabet of a regular class graph consists of the *terminal classes* (those concrete classes with a single outgoing syntax edge as their only part). Vertices representing non-terminal concrete classes are concatenation ($\cdot$) operators and vertices representing abstract classes are union ($+$) operators. The closure ($*$) operator can be expressed using a cycle combining vertices representing abstract and concrete classes. For convenience, we introduce a symbol (see Figure 7) for the $*$ operator and the symbol, λ, for the terminal class with the "empty string" as the target of its syntax edge.

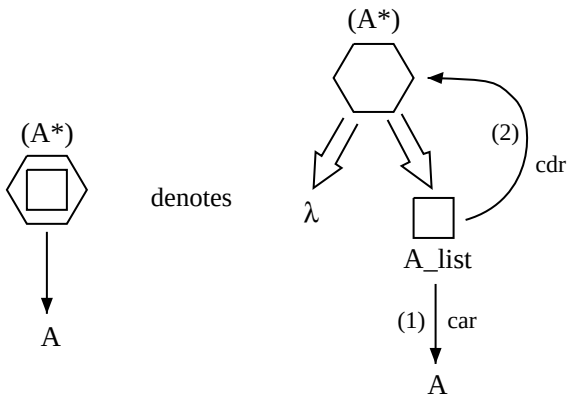
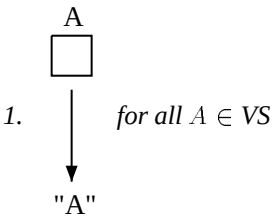
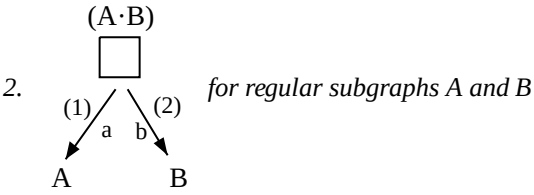
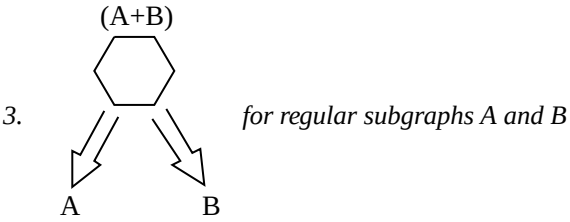


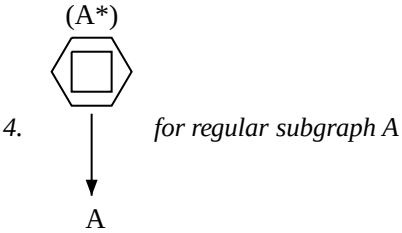
FIG. 7. Closure operator

Definition. *Let VS be the set of all syntax vertices. Then the* **regular** *class graphs and their bijection with the regular expressions are defined recursively as follows:*









Since there is a bijection between the regular expressions and the regular class graphs, we sometimes denote a regular class graph by its corresponding regular expression in the following discussion.

5.2.2. Axiom systems for regular expressions It is well known that using substitution as the only rule of inference, there is no finite set of equations over the regular expressions that allows the derivation of a regular expression, α, from a regular expression α' iff α and α' define the same language. That is, there is no finite complete set of axioms in the algebra of regular expressions [32]. However, the following system, $\mathcal{F}$, due to Salomaa [32] is complete if an additional rule of inference is allowed:

$$(\alpha + (\beta + \gamma)) = ((\alpha + \beta) + \gamma)$$
$$(\alpha \cdot (\beta \cdot \gamma)) = ((\alpha \cdot \beta) \cdot \gamma)$$
$$(\alpha + \beta) = (\beta + \alpha)$$
$$(\alpha \cdot (\beta + \gamma)) = ((\alpha \cdot \beta) + (\alpha \cdot \gamma))$$
$$((\alpha + \beta) \cdot \gamma) = ((\alpha \cdot \gamma) + (\beta \cdot \gamma))$$
$$(\alpha + \alpha) = \alpha$$
$$(\alpha \cdot \lambda) = \alpha$$
$$(\alpha \cdot \phi) = \phi$$
$$(\alpha + \phi) = \alpha$$
$$(\alpha*) = (\lambda + (\alpha \cdot (\alpha*)))$$
$$(\alpha*) = ((\lambda + \alpha)*)$$

The additional rule of inference is *solution of equations*: If β does not possess the "empty word property" then the equation $\alpha = (\gamma \cdot (\beta*))$ may be inferred from the equation $\alpha = ((\alpha \cdot \beta) + \gamma)$.

Definition. *A regular expression, α, (or its corresponding regular class graph) has the* **empty word property** *(e.w.p.) iff one of the following holds:*

1. $\alpha = \lambda$
2. $\alpha = (\beta*)$ for any β
3. $\alpha = (\beta + \gamma)$ where β or γ has the e.w.p.
4. $\alpha = (\beta \cdot \gamma)$ where β and γ have the e.w.p.

5.2.3. Completeness proof We show that for each equation over the regular expressions which is a substitution instance of an axiom in the complete system $\mathcal{F}$, there is a sequence of primitive transformations to transform the regular class graph corresponding to the left hand side of the equation to the class graph corresponding to the right hand side. Since every primitive transformation has an inverse, it is also possible to transform the right hand side to the left hand side. In other words, if it is possible to substitute a regular expression, α, for a regular expression α' in $\mathcal{F}$, then it is possible to transform the regular class graph, α' to α with the primitive transformations. Any regular expression, α, can be derived from any language-equivalent regular expression, α', in $\mathcal{F}$, so it must be possible to transform any regular class graph to any language-equivalent regular class graph by a sequence of primitive transformations if the following meta-transformation (Figure 8), corresponding to the extra rule of inference, solution of equations, is allowed:

- If a regular class graph, ϕ, contains a subgraph, $((\alpha \cdot \beta) + \gamma)$, which was obtained by a sequence of primitive transformations from α, and β does not possess the empty word property, then $((\alpha \cdot \beta) + \gamma)$ may be replaced with the subgraph $(\gamma \cdot (\beta*))$.

The meta-transformation may prove difficult to apply in practice and is not considered one of the primitives. It is included here only to demonstrate that the primitive transformations themselves form a system as complete as any known system for regular expressions.

Appendix C gives sequences of primitive transformations that correspond to each of the equations in the axiom system, $\mathcal{F}$, except for the equations $(\alpha \cdot \phi) = \phi$ and $(\alpha + \phi) = \alpha$ which are not applicable since the regular class graphs as defined here do not contain the empty language, ϕ.

6. CG Program Transformations

In this section, code update rules for programs written in the CG language are given for each of the primitive language preserving transformations. The rules are relatively straight forward since CG is untyped, and since the objects in a CG program always form a tree at runtime. Some of the code transformations require that an object have access to its "container" (parent) object and may involve adding code to the container class. For example, when a part is unnested (moved up the part-of hierarchy) instances of the class which originally had the part must access it indirectly through their containers. In CG there is a built-in method called `container` that provides the required access. In "real" languages, an instance variable may be added where necessary to provide a link to an object's container. Still, there are additional complications when the objects don't form a tree. When a class graph is used to define the class structure for a program written in a typical object-oriented language, a construction edge from some vertex, A, to another vertex, B, implies that every A object has a B object as a part, but not the converse; every B object is not necessarily a part of an A object. In general, there may be no suitable container class for a code transformation, and if there is a suitable class it cannot in general be identified without examining the existing code. The code transformations presented below, while correct for CG programs, are intended to be used with human guidance in the general case. The container classes are specified manually, and code must be transformed manually if a container class is required by a code transformation rule and none is available. For strongly typed languages there are the complications discussed in [7].

6.1. Transformation rules

6.1.1. The object-preserving transformations For the CG language, there are no code updates required for any object-preserving transformation. For other untyped languages, such as CLOS, the only object-preserving transformation that requires a code update is *deletion of useless alternation*. Note that the "useless" designation is only relevant from a data modeling point of view, since the class may have important methods attached. If the class is deleted the functionality of the methods attached to the class must be preserved. Each method can be copied to each of the immediate subclasses that does not override it. Now every object will respond to messages in the same way after the "useless" class is deleted. In CG, there are never any methods to copy since methods may only be attached to concrete classes.

6.1.2. Renaming of vertices and edges No code updates are required when vertices are renamed. In the standard interpretation, renaming an edge corresponds to changing the identifier for an instance variable. When an edge, $(v \xrightarrow{l} w)$, is renamed to $(v \xrightarrow{l'} w)$, the identifier l' is substituted for the identifier, l, in the methods of class v. Since CG provides strong data encapsulation, there is no need to make substitutions in methods of any other classes.

6.1.3. Nesting of parts If a class, A, has outgoing attribute edges, $(A \xrightarrow{b} B)$ and $(A \xrightarrow{c} C)$, and the edge $(A \xrightarrow{c} C)$ is replaced by an edge $(B \xrightarrow{c} C)$ (the c part of A is nested under its b part), then in methods attached to class A, the c part must be accessed indirectly through its b part. An accessor method to return the c part is added to class B, and the identifier, `c`, is replaced by `b <- c()` in methods of class A. If methods of class C access A objects through the `container` operator, the access must now be indirect through the intermediate B object. We add an accessor method to class B for its container, and in methods of class C the expression `container` is replaced by the expression `container <- container()`.

6.1.4. Unnesting of parts If a class, A, has an outgoing attribute edge, $(A \xrightarrow{b} B)$ to a class B which has an outgoing attribute edge, $(B \xrightarrow{c} C)$, to class C, and the edge $(B \xrightarrow{c} C)$ is replaced by the edge, $(A \xrightarrow{c} C)$, (the c part is unnested from under the b part of class A) then in methods attached to class B, the c part must be accessed indirectly through its parent A object. Similarly, in methods of class C that access B objects through the `container` operator, the access must now be indirect through its parent A object. Accessor methods are added to class A to return the b and c parts. In methods of class B, the expression `c` is replaced by the expression `container <- c()` and in methods of class C the expression `container` is replaced by `container <- b()`.

6.1.5. Addition of lambda parts Adding a part does not require any change in the code.

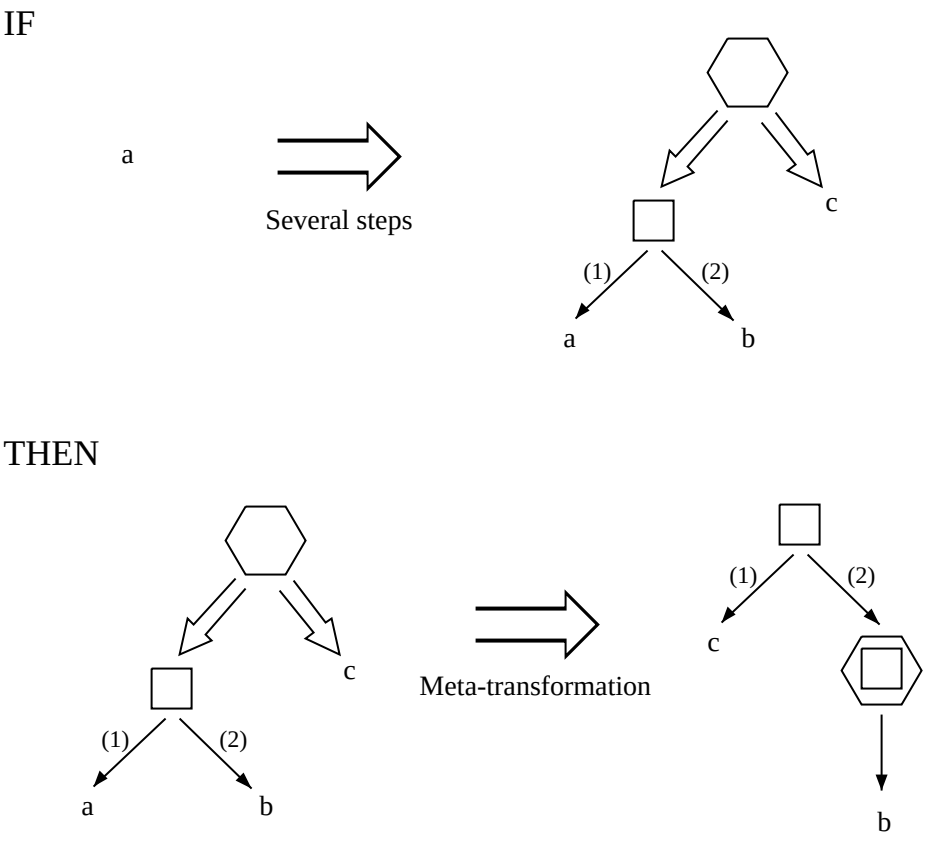


FIG. 8. Solution of equations

6.1.6. Deletion of lambda part If a class, A, has an attribute edge, $(A \xrightarrow{b} B)$, to a "lambda" class B and the edge is deleted, class A must supply any functionality that was formerly delegated to class B. Note that since B is a lambda class, none of its methods have access to any objects other than `self` and `container`. Each method of class B is copied unchanged to class A except that the expression `container` is replaced with `self` in the copied methods. In class A's original methods, the expression `b` is replaced with the expression `self`.

6.1.7. Addition of lambda alternative When an inheritance edge, $(A \Longrightarrow B)$, is added from a class, A to a lambda class, B, by addition of lambda alternative, B objects may be interspersed in a list of A objects. Any message received by such a B object should be passed to the A object that has been displaced. For each method attached to any subclass of A a corresponding method is generated for class B which simply delegates to the next object in the list. We also add to each of the original A classes a method called `this` which returns `self`. To class B we add a `this` method that returns `container <- this()`. In each of the original A methods the expression `container` is replaced by `container <- this()` so that these expressions will have the same objects as their values as if the B objects were not present in the list.

6.1.8. Deletion of lambda alternative Deletion of a lambda alternative does not require any change in the code.

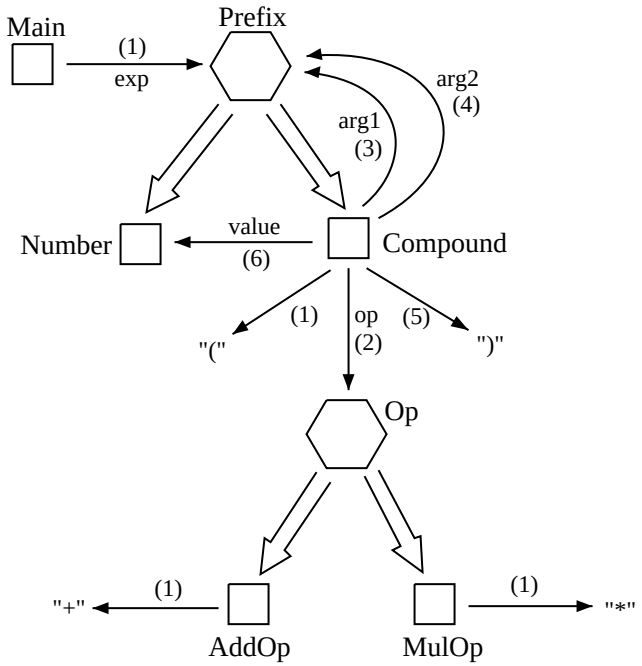
6.1.9. Insertion of singleton construction When a new concrete class, A, with an attribute edge to a class, B, is added by insertion of singleton construction, a method which simply delegates to its B part is added to class A for each method in class B. An accessor is added to A for its container, and in the methods of class B the expression `container` is replaced by `container <- container()`.

6.1.10. Deletion of singleton construction When a concrete class, A, with an attribute edge, $(A \xrightarrow{b} B)$, to class B is deleted by deletion of singleton construction, each method in class A is copied to class B with substitution of the expression `self` for `b`. In the original methods of class B the expression `container` is replaced by `self`.

6.1.11. Attribute to subclass When a concrete class is transformed into an abstract class by attribute to subclass, its methods are simply copied to each of its new subclasses.

6.1.12. Subclass to attribute When an abstract class, A, gains an attribute, l by subclass to attribute, each of its subclasses, B, must be a singleton construction whose only

outgoing edge, $(B \xrightarrow{l} C)$, has label l. The elimination of the subclasses is handled as for deletion of singleton construction (above), except that an additional argument is added to each of the methods copied from class B to C, and the copied methods are modified to access any parts inherited in B (from A) indirectly through the extra argument. Accessor methods are added to class A for each of its parts, and for each method copied from B to C, a corresponding method is added to class A which simply delegates to its l part, passing along whatever arguments it received plus `self` for the actual value of the extra argument.

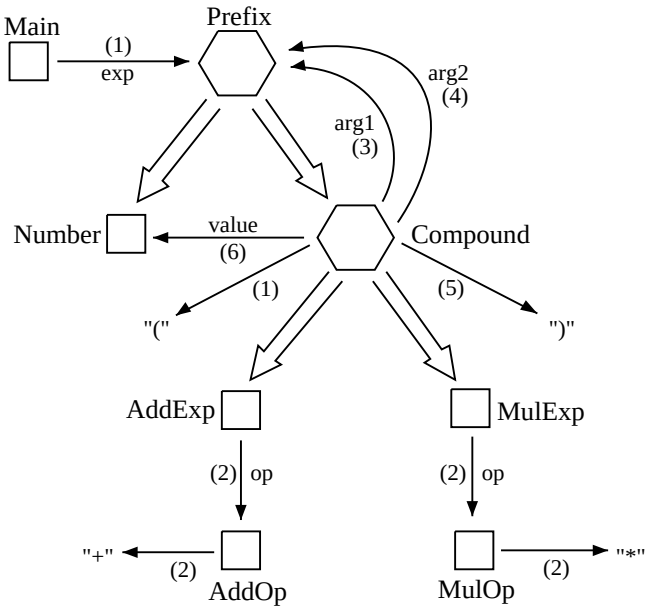



```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

Compound : eval ()
{ op <- apply(arg1, arg2, value) }

AddOp : apply (a1, a2, v)
{
  v <- assign(a1 <- eval())
    <- add(a2 <- eval())
}

MulOp : apply (a1, a2, v)
{
  v <- assign(a1 <- eval())
    <- mul(a2 <- eval())
}
```

FIG. 9. Program C

6.2. Examples of CG program transformations

Example 3. *Figures 9-11 show how yet another version of the prefix expression evaluator (Figure 9) is transformed when its class structure evolves first by transforming the* `op` *attribute of class* `Compound` *to subclasses followed by deletion of the useless alternation vertex* `Op` *(Figure 10) and then by deleting the singleton construction vertices* `MulExp`

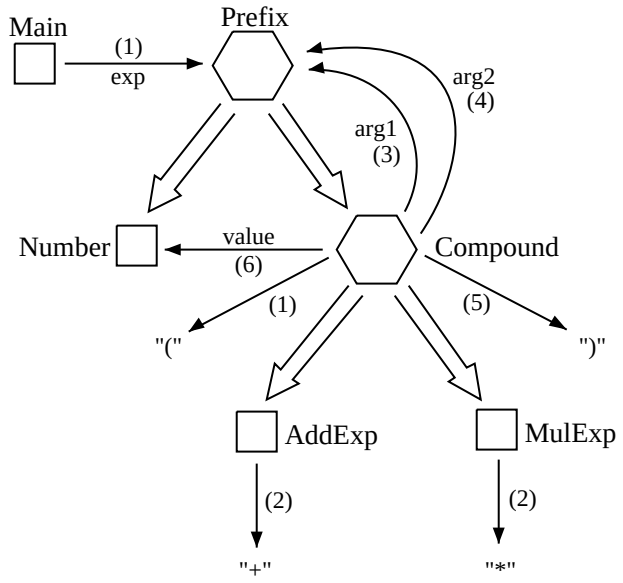



```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

MulExp : eval ()
{ op <- apply(arg1, arg2, value) }

AddExp : eval ()
{ op <- apply(arg1, arg2, value) }

AddOp : apply (a1, a2, v)
{
  v <- assign(a1 <- eval())
    <- add(a2 <- eval())
}

MulOp : apply (a1, a2, v)
{
  v <- assign(a1 <- eval())
    <- mul(a2 <- eval())
}
```

FIG. 10. Program D

and `AddExp` *followed by renaming of the classes* `MulOp` *and* `AddOp` *to* `MulExp` *and* `AddExp`*, respectively (Figure 11).*

Example 4. Figure 12 shows a CG program to calculate the total weight of all the bricks in a pile. If the class graph evolves by addition of lambda alternative to allow balloons to be interspersed with the bricks the code to calculate the total weight of the bricks is updated as shown in Figure 13.




```
Main : main ()
{ exp <- eval() <- print() }

Number : eval ()
{ self }

MulExp : eval ()
{ self <- apply(arg1, arg2, value) }

AddExp : eval ()
{ self <- apply(arg1, arg2, value) }

AddExp : apply (a1, a2, v)
{
  v <- assign(a1 <- eval())
    <- add(a2 <- eval())
}

MulExp : apply (a1, a2, v)
{
  v <- assign(a1 <- eval())
    <- mul(a2 <- eval())
}
```

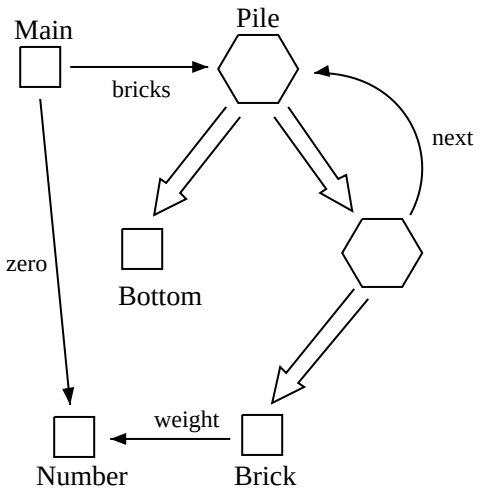
FIG. 11. Program E

7. Practicality of the approach

7.1. Limitations of the Data Model

The data model used here is a formal mathematical model (see Appendix A), that was chosen to provide programming language independence and a sound theoretical basis for the methodology. It is a "low-level" model that starts with only those relationships that are directly supported by most most object-oriented programming languages: part-of (data members) and kind-of (subclassing). Concrete syntax and ordering of parts are added to relate the semantics of different class structures.

It is expected that the approach will often be applied in the same way it was developed. That is, a class structure without concrete syntax or part ordering will be embedded in a grammar for the purpose of maintenance. The grammatical requirements of the model should not restrict its usefulness. In many cases, it may even be possible to recognize class transformations as "language-preserving" outside the context of a grammar. Still, the concrete syntax and ordering of

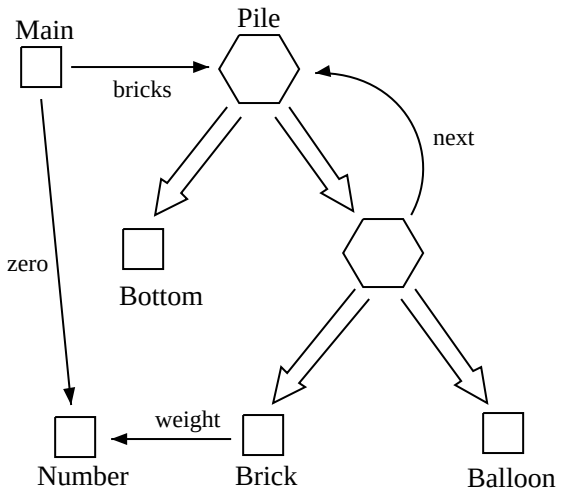



```
Main : main ()
{ bricks <- last() <- weight() <- print() }

Brick : last ()
{ next <- last() }

Bottom : last ()
{ container }

Brick : weight ()
{ container <- weight() <- add(weight) }

Main : weight ()
{ zero }
```

FIG. 12. Program to calculate weight of brick pile

parts is a valuable means of documenting the nature of class transformations.

The main constraints in our model are the lack of inheritance from concrete classes and the inability to override (or shadow) data members in subclasses. We also disallow name clashes due to multiple inheritance. None of these constraints pose a major obstacle to the use of the approach.




```
Main : main ()
{ bricks <- last() <- weight() <- print() }

Main : weight ()
{ zero }

Brick : last ()
{ next <- last() }

Brick : weight ()
{
  container <- this()
     <- weight() <- add(weight)
}

Brick : this ()
{ self }

Bottom : last ()
{ container <- this() }

Balloon : this ()
{ container <- this() }

Balloon : last ()
{ next <- last() }

Balloon : weight ()
{ next <- weight() }
```

FIG. 13. After adding balloons to the pile

A class organization with inheritance from concrete classes may be easily (and automatically) restructured to eliminate this kind of inheritance. The concrete class is replaced by an abstract class in the inheritance hierarchy, the concrete class is made a subclass of the new abstract class, and all methods of the concrete class are moved to the new abstract class. There are no code transformations required.

In the author's opinion, shadowing data members in subclasses is a dangerous and generally undesirable technique. In any case, it can be simulated by implementing the data member as a method, and overriding in subclasses.

Any language, or model, that allows multiple inheritance must somehow cope with potential name clashes. For example, in Eiffel, there is a mechanism to rename inherited data members. In C++, names must be qualified by using the class scope resolution operator. These design decisions have only a minor effect on the mechanics of code transformations. By requiring unique names, we have, in essence, adopted the C++ approach. If we consider instance variables to have the class name where they are defined as an implicit prefix, we get unique names.

Our low-level model is probably unsuitable for high-level analysis and design. However, models in common use, including the new Unified Modeling Language, may be mapped to the CG model in the same way that they are mapped to programming languages. A CASE tool supporting such models could incorporate support for maintenance of user written code based on the mapping to low-level constructs.

7.2. Limitations of the Language Model

The limitations of the CG language model include:

- No conditional expressions
- No looping constructs
- No inheritance of methods
- No programmer control over encapsulation
- No true dynamic object construction
- Object structure at run time must be a tree
- The language is untyped
- No non-object primitives (e.g. integer, character, etc.)

Most of these language features were left out of the model merely to simplify the discussion, and to allow us to focus on those features at the core of object-oriented programming: data encapsulation, message passing, late binding, and polymorphism. We are currently implementing a system that will support all of the features mentioned above except static typing. Our implementation will support C++, but circumvents the type system by supplying an `Object` class from which every other class inherits. We define a default, *message not understood*, method for class `Object` corresponding to each method in every other class. All member functions and parameters are declared to have type `Object`. We provide "wrapper" classes for the non-object types. In the future, we will eliminate the `Object` class, and modify our transformations to satisfy the C++ type system. Many of the issues

relevant to type system support in program transformations have already been investigated [7].

The addition of conditionals and loops has had no impact on the transformation rules. Inheritance of methods has very little effect. In those cases were it is important, a method attached to an abstract class can be copied to each of the subclasses that does not override it, and then eliminated. If a method in a subclass calls this method using the class scope resolution operator, the code can be inlined. Later, identical methods may be abstracted to common superclasses in the same way that data members are abstracted.

The program transformations assume that all methods are public, and all data members are protected. If stronger encapsulation is present, and a program transformation requires access which is not allowed, we simply weaken the encapsulation (with appropriate warnings and user interaction). In C++ the only encapsulation weaker than what we have assumed, is the use of public data members. We consider this very poor programming practice and our system will not support it.

The problem with object structures that are not trees, in general, and of dynamic object creation, in particular, is that objects may not have a suitable container object for transformations that require it. As noted in Section 6, user interaction is necessary in such cases.

7.3. Reorganizations that are not language preserving

The most commonly occurring reorganizations that are not language preserving are those that add new classes, or new attributes to existing classes. These *object-extending* class transformations are much easier to manage than the language-preserving transformations described here. A complete set of primitive object-extending transformations, and corresponding code update rules is described in [7]. The maintenance techniques for language-preserving transformations are intended to augment, rather than replace, existing techniques to allow for automatic maintenance over a much larger set of class reorganizations.

7.4. Practical experience

The original motivation for this work was a major revision of the Demeter system's class graph, which required the entire system to be manually ported to the new environment by rewriting all the code. This was true even though the languages defined by the class graphs were nearly identical, and the functionality of the programs comprising the system was unchanged.

The methodology presented in this paper has been applied, by hand, to parts of the system that motivated the research. The Demeter System [17, 33] originally used a notation based on grammars and later changed to a graph based notation. When the notation was changed the class structure was reorganized to properly model the new perspective. For example, a small portion of the system's original class structure and the corresponding portion of the new structure are shown in Figure 14.

The strategy suggested in Section 8.2 was used to find a sequence of primitives to accomplish the overall class structure transformation. This strategy proved effective for the test case with approximately 40 classes in each structure. A sequence of primitives was found in approximately 2 hours.

Next, portions of the CLOS code used to implement the original system were modified according to the rules in Section 6 (adding instance variables to link objects to their "containers" where necessary), and the correctness of the new code was verified.

The same methodology has been used successfully to guide the evolution of more recent versions of the Demeter System which have been implemented in C++. In this case, the code transformation rules had to be augmented somewhat in order to satisfy the type system. For a more complete discussion of type system issues see [7].

8. Search algorithms

If the primitive language-preserving transformations are used to restructure the class organization of a CG program, the code may be automatically updated following the rules defined in Section 6. More generally, given an arbitrary CG program and a new language-equivalent class graph, we must be able to find a sequence of primitives that produces the given transformation in order to apply the code transformation rules.

8.1. Regular languages

Since the primitive transformations are not complete for regular class graphs without the addition of a meta-transformation, it is not always possible to reduce an arbitrary language-preserving transformation over the regular class graphs to a sequence of primitives. However, there is an algorithm to perform the reduction to a sequence of primitives and meta-transformations. Manual code updates must then be performed only for the meta-transformations.

The proof that Salomaa's axiom system for the regular expressions is complete [32] is constructive in the sense that for any valid equation $X = Y$, over the regular expressions it gives a method to construct its proof. To reduce an arbitrary language-preserving transformation over the regular class graphs to a sequence of primitives we first construct the proof that the corresponding regular expressions are equivalent. Each substitution in the proof is mapped to a sequence of primitives as defined in Section 5.2.3.. Each solution of equations is mapped to the meta-transformation defined in Section 5.2.3..

8.2. Context free languages

There can be no algorithm guaranteed to reduce an ar-

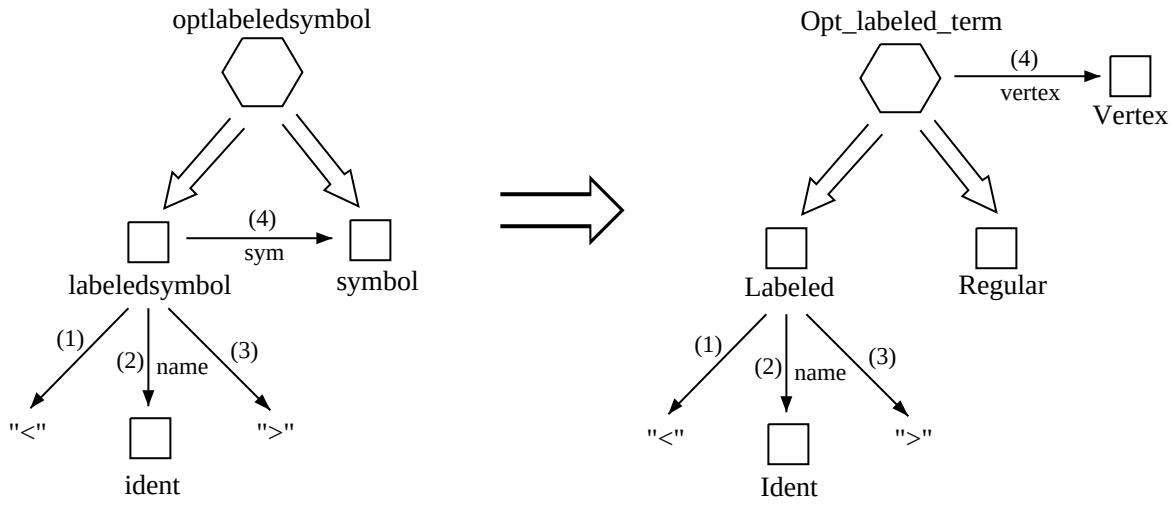


FIG. 14. Reorganization of the Demeter System class structure

bitrary language-preserving transformation over the class graphs to a sequence of primitives. Even if the set of primitive transformations were complete, such an algorithm would be impossible since equivalence of context-free languages is undecidable, and the class graphs define context-free languages[4]. Nevertheless, it is reasonable to expect that searches for sequences of primitives will often terminate successfully since the primitives are designed to represent the kinds of transformations that are likely to arise in practice.

The search problem may be viewed in terms of the classic state-space search paradigm as defined in the literature of artificial intelligence. Given an initial state, S, a set of operators on states, O, and a set of goal states, G, the state-space is defined as a directed graph where each node represents a state and each arc represents an operation. The problem is to find a path from the initial state to a goal state. Normally, the graph is not made explicit except for the solution path.

In our case, the initial state is a class graph, the operators are the primitive language-preserving transformations, and the only goal state is a language-equivalent class graph. Alternatively, we may consider the set of goal states to be the set of all class graphs which are object-equivalent to a given language-equivalent class graph since we already have efficient algorithms for checking object-equivalence and reducing an object-preserving transformation to a sequence of primitives.

State-space search has been heavily investigated in AI, and sophisticated systems have been developed for evaluating states and choosing the next operation to apply in various domains. A detailed algorithm of this sort is beyond the scope of the current work and is left for future research. However, a simple search strategy might proceed as follows: The state space is searched in depth-first order (with backtracking) and operators are applied to vertices in the class graph in breadth-first order starting with the `Main` class. The `Main` vertex of the initial class graph is brought into congruence with the `Main` vertex of the goal state by applying operators (primitive transformations) until the vertices have the same types and numbers of outgoing edges, outgoing construction edges have the same labels, and outgoing syntax edges have the same targets. Next, the target of each construction and alternation edge is brought into congruence with the corresponding class in the goal state in the same manner, and the process continues until the target is reached or the depth in the state space exceeds some specified value.

An important heuristic which can be used to improve the search performance is to use the names of classes and labels of edges to guide the search. If, for example, a vertex must have an outgoing construction edge with label l to a vertex labeled V, we first check to see if there is already an outgoing edge with label l. If two analogous classes have parts with the same names, we guess that the parts are also analogous. Otherwise, we check if some other vertex has an outgoing edge with label l and target V that can be brought into the proper position by nesting and unnesting of parts. If neither condition is met we look for a vertex labeled V and finally for an edge with label l.

This strategy is useful if the class graph changes gradually during the evolutionary process, since most classes and parts will retain their original names. It is also useful if the designers use names consistently when reorganizing the class structure. Finally, if a class graph has changed dramatically it may be easy for a human designer familiar with the application domain to supply a mapping between classes with analogous roles by manually renaming parts and classes before starting the search. For the human designer, giving a partial analogy by renaming the parts and classes is the "easy" part – elaborating the analogy by finding the primitive transformations and then updating all of the code is the "hard" part. For the machine, the reverse is true; thus, the machine complements the abilities of the human designer when this strategy is employed.

The concrete syntax may be used as a further guide of the search or to prune nodes in the state space if we note that it is not possible to find a solution by bringing two vertices

into congruence that have different sets of reachable syntax vertices.

9. Related Work

9.1. Software Refactoring

In their software refactory project, Opdyke and Johnson [28, 27, 29, 19] have worked on building a tool to support various aspects of object-oriented program transformation, including the maintenance of behavioral consistency during schema evolution. Many of the code transformation issues they address are similar to the issues addressed here, and their solutions are also similar in some cases. In particular, they consider reorganization of "aggregate/component" hierarchies, and conversion between aggregation and inheritance. The problems and solutions they present are quite similar to those presented here for nesting/unnesting of parts and subclass-to-attribute/attribute-to-subclass conversions, respectively.

The work of Opdyke and Johnson differs in several ways from the work presented in this paper. Perhaps the most important is the theoretical basis of the current work in formal languages. In our case, the program transformation space is clearly and concisely defined. Furthermore, language-equivalent class graphs guarantee that programs written in the CG language will accept the same inputs, and that their run-time objects have the same textual descriptions. This is the answer to the important question: "Why is it reasonable to expect the existence of code that will make a system of transformed objects behave in a manner functionally equivalent to the original system?".

The alternative answer is that the transformation was accomplished via a sequence of primitives for which correct code transformations are known. Opdyke and Johnson rely solely on this second justification, and require that users of their system directly apply primitives (or compositions of primitives already known to the system). In the system we envision, users need not even be aware of the existence of the primitive transformations. In the context of reuse (as opposed to evolution), the users of Opdyke and Johnson's system would have to perform a search for a sequence of primitives to transform the class structure of the existing code to the class structure where it is to be reused. In the system we envision, there is a search engine to perform this task automatically.

9.2. Structure Mapping Theory

In structure mapping theory [14, 15] knowledge is represented as propositional networks comprising object nodes and predicates (attributes and relations). An analogy maps object nodes from the base domain to object nodes in the target domain. Generally, there is a 1-1 mapping between nodes in the base and target domains. Each pair of corresponding object nodes in the mapping is part of the analogy: "the target is like the base". The analogy is applied by using mapping rules, based on the principle of *systematicity*, to determine which predicates should be brought from the base domain to the target domain. The selected predicates are carried over using the node substitutions indicated in the object mapping.

A sequence of primitive transformations where each primitive only renames a vertex in a class graph would be equivalent to an analogy as defined by structure mapping theory. However, the primitive transformations can be more expressive since they may include changes in certain *structural relations* (e.g. part-of, kind-of) as part of the analogy. For example, in the base domain a class `Human` might have an attribute (part) called `Gender`, with possible values (kinds) `Male` or `Female`. In the target domain an analogous structure might have a class `Person` with subclasses (kinds) `Man` and `Woman`. The simple mapping $(Human \rightarrow Person, Male \rightarrow Man, Female \rightarrow Woman)$ does not properly express the analogy. Instead, the relationship between `Person` and `Human` must be qualified, as in: "a `Person` is like a `Human` where the attribute `Gender` is expressed by subclassing". Gentner's structure mapping theory is not powerful enough to express such a qualified structural analogy, but this can be expressed by a primitive transformation, say "attribute-to-subclass".

In our work, application of the analogy involves bringing relations, in the form of program code, from the base domain over to the target domain. As in structure mapping theory, the rules depend only on syntactic properties and not on an understanding of the contents of the domains. Therefore, the code is brought over with little modification.

Structure mapping theory says nothing about how an analogy, "the *target* is like the *base*", is broken down into a mapping of nodes in the base to nodes in the target. In our case, a search is performed to find a sequence of primitive transformations that would convert the base structure to the target structure.

9.3. Analogical Program Synthesis Guided by Correctness Proofs

Ulrich and Moll [38] have used correctness proofs to guide the formation of analogies and the construction of analogous programs. Each line in the proof of a program written for the base domain is mapped into a statement in the target domain. Terms and relationships in the target domain are substituted for terms and relationships in the original proof. As the process is carried out, the original program is modified by the same substitutions. This process produces a new program and its correctness proof at the same time.

Dershowitz and Manna [13] used a similar approach to automatically modify programs. They formulate an analogy as a set of substitutions that yield a specification of the desired program when applied to the specification of an analogous

program. The specifications, including input specifications, are given for both programs in a high-level assertion language. In our case, the known CG program contains its own input specification in the form of a class graph.

An important aspect of a program specification is the inclusion of *invariant assertions* which are utilized in correctness proofs. Transformations are applied to all assertions as well as to program code. The transformed assertions can be used to obtain verification conditions for the new program. In our case, it is the language defined by the class graphs that remains invariant. Correctness is guaranteed by the correctness of the primitive transformations.

10. Conclusions

By extending a typical graph based data model to include concrete syntax, we have produced a new model that can be used to simultaneously define both a class structure and a language for describing instances of the classes textually. When the extended data model is incorporated into a programming environment, we get programs that define a language for describing their run-time objects and are self documenting as to their legal inputs. The result is a novel framework for dealing with object restructuring with a theoretical basis in formal languages.

The methods described for maintaining behavioral consistency of evolving systems have been successfully applied by hand to the development of the Demeter System. An automated prototype is currently in the planning stage.

Acknowledgments

The author would like to thank the anonymous referees of TOPLAS and TAPOS for their many helpful comments on earlier versions of this article.

Notes

1. More generally, if two class graphs define a common sub-language (i.e. the intersection of the two languages is not empty) then a program written for one of the class graphs could be automatically transformed into a program for the other in such a way that the behavior of the system is preserved for any input in the sub-language.
2. The required accessor methods for class `Compound` are not shown.
3. There may be more than one container for each object.
4. See Table 1.

References

[1] Serge Abiteboul and Richard Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

[2] H. Ait-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[3] Jay Banerjee, Won Kim, Hyong-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 311–322. ACM, ACM Press, December 1987. SIGMOD Record, Vol.16, No.3.

[4] Gilles Barbedette. Schema modifications in the $lispo_2$ persistent object-oriented language. In Pierre America, editor, *European Conference on Object-Oriented Programming*, pages 77–96, Geneva, Switzerland, July 1991. Springer Verlag, Lecture Notes in Computer Science.

[5] Paul L. Bergstein. Object-preserving class transformations. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 299–313, Phoenix, Arizona, 1991. ACM Press.

[6] Paul L. Bergstein. *Managing the Evolution of Object-oriented Systems*. PhD thesis, Northeastern University, Boston, Massachusetts, June 1994.

[7] Paul L. Bergstein and Walter L. Hürsch. Maintaining behavioral consistency during schema evolution. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 176–193, Kanazawa, Japan, November 1993. JSSST, Springer Verlag, Lecture Notes in Computer Science. Also available as Northeastern University, College of Computer Science technical report number NU-CCS-93-04.

[8] Elisa Bertino. A view mechanism for object-oriented databases. In *International Conference on Extending Database Technology*, pages 136–151, Vienna, Austria, 1992.

[9] Alexander Borgida, Tom Mitchell, and Keith Williamson. Learning improved integrity constraints and schemas from exceptions in data and knowledge bases. In Michael L. Brodie and John Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 259–286. Springer Verlag, 1986.

[10] Eduardo Casais. *Managing evolution in object-oriented environments: an algorithmic approach*. PhD thesis, University of Geneva, Geneva, Switzerland, May 1991. Thesis no. 369.

[11] Alberto Coen-Porisini, Luigi Lavazza, and Roberto Zicari. Updating the schema of an object-oriented database. *Quarterly Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 14(2):33–37, June 1991. Special Issue on Foundations of object-Oriented Database Systems.

[12] Christine Delcourt and Roberto Zicari. The design of an integrity consistency checker (icc) for an object oriented database system. In Pierre America, editor, *European Conference on Object-Oriented Programming*, pages 97–117, Geneva, Switzerland, July 1991. Springer Verlag, Lecture Notes in Computer Science.

[13] Nachum Dershowitz and Zohar Manna. The evolution of programs: Automatic program modification. *IEEE Transactions on Software Engineering*, SE-3(6):377–385, November 1977.

[14] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7:155–170, 1983.

[15] Dedre Gentner and Cecile Toupin. Systematicity and surface similarity in the development of analogy. *Cognitive Science*, 10:277–300, 1986.

[16] R.B. Hull and C.K. Yap. The format model: A theory of data organization. *Journal of the Association for Computing Machinery*, 31(3):518–537, July 1984.

[17] Walter L. Hürsch, Linda M. Seiter, and Cun Xiao. In any CASE: Demeter. *The American Programmer*, 4(10):46–56, October 1991.

[18] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[19] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 264–278, Kanazawa, Japan, November 1993. JSSST, Springer Verlag, Lecture Notes in Computer Science.

[20] G.M. Kuper and M.Y. Vardi. The logical data model. In *Principles of Database Systems*, pages 86–96. ACM, 1984.

[21] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In Norman Meyrowitz, editor, *Proceedings OOPSLA ECOOP '90*, pages 67–76, Ottawa, Canada, October 1990. ACM, ACM Press. Special Issue of SIGPLAN Notices, Vol.25, No.10.

[22] Qing Li and Dennis McLeod. Conceptual database evolution through learning in object databases. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):205–224, 1994.

[23] Karl J. Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(2):185–212, 1988.

[24] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: Algorithms for object-oriented design. *Journal of Software Engineering*, 6(4):205–228, July 1991.

[25] Karl J. Lieberherr and Arthur J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August, September 1988. A shorter version of this paper was presented at the *10th International Conference on Software Engineering, Singapore, April 1988, IEEE Press*, pages 254-264.

[26] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall International, 1988.

[27] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing object-Oriented Application Frameworks*. PhD thesis, Computer Science Department, University of Illinois, May 1992.

[28] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications (SOOPA)*, pages 145–160, Poughkeepsie, NY, September 1990. ACM.

[29] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of CSC '93: The ACM 1993 Computer Science Conference*, February 1993.

[30] Jason D. Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In Norman Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 111–117, Orlando, Florida, December 1987. ACM, ACM Press. Special Issue of SIGPLAN Notices, Vol.22, No.12.

[31] B. Pernici, F. Barbic, M.G. Fugini, R. Maiocchi, J.R. Rames, and C. Rolland. C-TODOS: An automatic tool for office system conceptual design. *ACM Transactions on Office Information Systems*, 7(4):378–419, October 1989.

[32] Arto Salomaa. *Theory of Automata*. International series of monographs in pure and applied mathematics, v. 100. Pergamon Press, 1969.

[33] Ignacio Silva-Lepe, Walter Hürsch, and Greg Sullivan. A Report on Demeter/C++. *C++ Report*, pages 24–30, February 1994.

[34] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 483–495. ACM, ACM Press, September 1986.

[35] Richard Snodgrass. *The interface description language*. Computer Science Press, 1989.

[36] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational data bases. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[37] Dennis Tsichritzis and Frederick Lochovsky. *Data Models*. Software Series. Prentice-Hall, 1982.

[38] John Wade Ulrich and Robert Moll. Program synthesis by analogy. *SIGPLAN Notices*, (64):22–28, August 1977.

Appendix A

Data Model

A.1. Class graphs

The class graphs described in this section use three kinds of vertices to model abstract classes, instantiable classes, and concrete syntax. We also use three kinds of edges to model knows-of, kind-of, and has-syntax relationships. The knows-of relationship is a generalization of the aggregation relation which only describes physical containment.

The knows-of and has-syntax relations comprise what we call the "part-of" relation. For lack of a better term, we call the syntax and known collaborators of an object its "parts", although the parts need not be physical parts. For example, in our terminology a car is part of a wheel if the wheel knows about the car.

Definition. *A* **class graph**[1]*, ϕ, is a directed graph, $\phi = (V, VS, \Lambda; EC, EA, ES, Ord)$, with finitely many vertices V. VS is a set of strings called the syntax vertices. Λ is a finite set of labels. There are four defining relations: EC, EA, ES, Ord. EC is a ternary relation on $V \times V \times \Lambda$, called the (labeled) construction edges: $(v \xrightarrow{l} w) \in EC$ iff there is a construction edge with label l from v to w. EA is a binary relation on $V \times V$, called the alternation edges: $(v \Longrightarrow w) \in EA$ iff there is an alternation edge from v to w. ES is a binary relation on $V \times VS$ called the syntax edges: $(v \rightarrow w) \in ES$ iff there is a syntax edge from v to w. Ord : $(EC \cup ES) \rightarrow \mathcal{N}$ is a function that maps each construction and syntax edge to a natural number.*

Next the set of vertices is partitioned into two subclasses, called the construction and alternation vertices.

Definition.

- The **construction vertices** are defined by:
  $VC = \{v \mid v \in V, \forall w \in V : (v \Longrightarrow w) \notin EA\}$.
  In other words, the construction vertices have no outgoing alternation edges.
- The **alternation vertices** are defined by:
  $VA = \{v \mid v \in V, \exists w \in V : (v \Longrightarrow w) \in EA\}$.
  In other words, the alternation vertices have at least one outgoing alternation edge.

Sometimes, when we want to talk about the construction and alternation vertices of a class graph, it is more convenient to describe a class graph as a tuple which contains explicit references to *VC* and *VA*: $\phi = (VC, VA, VS, \Lambda; EC, EA, ES, Ord)$.

In standard object-oriented terminology we describe here the accepted programming rule: "Inherit only from abstract classes" [18]. This rule can be exploited to derive an analogy between class graphs and grammars.

We use the following graphical notation, based on [36], for drawing class graphs: squares for construction vertices, hexagons for alternation vertices, quoted strings for syntax vertices, thin arrows for construction and syntax edges, and wide arrows for alternation edges.

Example 1. For further illustration we give the components of the formal definition, for the class graph, ϕ_1, of

TABLE 1. Standard interpretation of class graphs

Graph	Object-oriented Design	Context Free Language
Vertex	**Class**	**Symbol**
construction	instantiable class with members defined by construction edges (including "inherited" edges)	Concatenation of languages defined by construction and syntax edges (including "inherited" edges)
alternation	abstract class with subclasses defined by alternation edges	union of languages defined by alternation edges
syntax	no meaning	terminal
Edge	**Class Relationship**	**Operator**
construction	part-of relationship, "uses", "knows", — labels are part names	concatenation — numbers define order
alternation	inheritance relationship, specialization, classification	union
syntax	no meaning	concatenation — numbers define order

Figure 1:

$$
\begin{aligned}
VC &= \{Number, AddExp, MulExp\} \\
VA &= \{Prefix, Compound\} \\
VS &= \{\text{“)”}, \text{“(”}, \text{“+”}, \text{“*”}, [0-9]+\} \\
\Lambda &= \{num, arg1, arg2\} \\
EC &= \{(Compound \xrightarrow{arg1} Prefix), \\
&\quad (Compound \xrightarrow{arg2} Prefix)\} \\
EA &= \{(Prefix \Longrightarrow Number), \\
&\quad (Prefix \Longrightarrow Compound), \\
&\quad (Compound \Longrightarrow AddExp), \\
&\quad (Compound \Longrightarrow MulExp)\} \\
ES &= \{(Compound \rightarrow \text{“(”}), (Compound \rightarrow \text{“)”}), \\
&\quad (Number \rightarrow [0-9]+), (AddExp \rightarrow \text{“+”}), \\
&\quad (MulExp \rightarrow \text{“*”})\} \\
Ord &= \{(Compound \xrightarrow{arg1} Prefix, 3), \\
&\quad (Compound \xrightarrow{arg2} Prefix, 4), \\
&\quad (Compound \rightarrow \text{“(”}, 1), \\
&\quad (Compound \rightarrow \text{“)”}, 5), (Number \rightarrow [0-9]+, 1), \\
&\quad (AddExp \rightarrow \text{“+”}, 2), (MulExp \rightarrow \text{“*”}, 2)\}
\end{aligned}
$$

The definition of *VC* implies that $EA \subseteq VA \times V$, since an alternation edge cannot start at a construction vertex. We use V_ϕ, VC_ϕ, VA_ϕ etc. to refer to the components of class graph ϕ.

When we draw a class graph, the vertices are labeled so that we can conveniently refer to particular vertices in our discussion. The standard interpretation implies that the labels on construction vertices are significant. Consider two isomorphic class graphs each with only a single construction vertex and no edges. If the construction vertex of one graph is labeled *Integer* and the vertex of the other graph is labeled *String*, then the two class graphs define different sets of objects in the standard interpretation. On the other hand, changing the labels of the alternation vertices (names of abstract classes in the standard interpretation) does not effect the defined objects. Therefore, we adopt the following convention for labeling the vertices of class graphs: Labels of alternation vertices are local to the class graph in which they occur; labels of construction vertices are global. That is, if two class graphs have construction vertices with the same label, it means that the *same vertex* (same class under the standard interpretation) belongs to both graphs. However, we may in general assume that different class graphs have disjoint sets of alternation vertices regardless of their labels.

The same semantics apply when we denote the sets of vertices in a class graph textually. The identifiers we use to denote alternation vertices are of local scope whereas the identifiers we use to denote construction vertices have global scope.

Later we give conditions which make a class graph into a legal class graph. The interpretation in Table 1 is only one possible interpretation which we call the standard interpretation. The motivation behind the abstract alternation/construction terminology is that there are several useful interpretations of class graphs. In one of those interpretations, a construction vertex is interpreted as an operation. We often use the standard interpretation to give intuitive explanations of relationships and algorithms.

Please note that the syntax for an alternation vertex/abstract class, although very natural from a graph-theoretic point of view, appears unnatural from the point of view of today's programming languages: In most programming languages which support the object-oriented paradigm, the inheritance relationships are described in the opposite way. Each class indicates from where it inherits. Of course, we can easily generate this information from class graphs, but we feel that the Demeter notation is easier to use for de-

sign purposes. One reason is that the design notation shows the immediate subclasses of a class and therefore promotes proper abstraction of common parts. Another reason is that a class does not contain information about where it inherits from and therefore the class can be easily reused in other contexts.

Definition. *In a class graph, $\phi = (V, VS, \Lambda; EC, EA, ES, Ord)$, a vertex $w \in V$ is* **alternation-reachable** *from vertex $v \in V$ (we write $v \stackrel{\star}{\Longrightarrow} w$):*

- via a path of length 0, if $v = w$
- via a path of length $n+1$, if $\exists u \in V$ such that $(v \Longrightarrow u) \in EA$ and $u \stackrel{\star}{\Longrightarrow} w$ via a path of length n.

In other words, the **alternation-reachable** relation is the reflexive, transitive closure of the *EA* relation.

In the standard interpretation, $(v \stackrel{\star}{\Longrightarrow} w)$ means that either w inherits from v or $w = v$.

Sometimes when we want to discuss the inheritance hierarchy, it is convenient to refer to the alternation subgraph of a class graph. The alternation subgraph contains all of the alternation vertices and alternation edges plus the construction vertices that have incoming alternation edges.

Definition. *The* **alternation subgraph** *of a class graph, $\phi = (VC, VA, \Lambda; EC, EA)$, is a directed acyclic graph (DAG), $G = (V', EA)$, where $V' = VA \cup \{v \in VC \mid \exists u : (u \Longrightarrow v) \in EA\}$.*

It is often helpful to think of each alternation vertex as representing a set of associated construction vertices. This set, $\mathcal{A}(v)$, consists of all the construction vertices which are alternation reachable from the vertex, v. If v is an alternation vertex with an incoming construction edge, $(u \stackrel{l}{\longrightarrow} v)$, the construction vertices in $\mathcal{A}(v)$ represent the concrete classes which might be used to instantiate the l part of u objects. If v has an outgoing construction edge, $(v \stackrel{l}{\longrightarrow} w)$, the construction vertices in $\mathcal{A}(v)$ represent the concrete classes which inherit the l part from v.

Definition. *The* **associated classes** *of a vertex, v, in a class graph, $\phi = (VC, VA, VS, \Lambda; EC, EA, ES, Ord)$, is the set of all construction vertices which are alternation-reachable from v:*

$$\mathcal{A}(v) = \{v' | v \stackrel{\star}{\Longrightarrow} v' \text{ and } v' \in VC\}$$

A.1.1. Legality conditions A legal class graph is a structure which satisfies three independent conditions.

Definition. *A class graph $\phi = (V, VS, \Lambda; EC, EA, ES, Ord)$ is* **legal** *if it satisfies the following three conditions:*

1. Cycle-free alternation condition:
   There are no cyclic alternation paths, i.e.,
   $\{(v, w) \mid v, w \in V, v \neq w, \text{ and } v \stackrel{\star}{\Longrightarrow} w \stackrel{\star}{\Longrightarrow} v\} = \emptyset$.
2. Unique labels condition:
   $\forall u, v, v', w, w' \in V, l \in \Lambda$ such that $(v \stackrel{\star}{\Longrightarrow} u)$, $(v' \stackrel{\star}{\Longrightarrow} u)$, and $(v, w) \neq (v', w')$:
   $\{(v \stackrel{l}{\longrightarrow} w), (v' \stackrel{l}{\longrightarrow} w')\} \not\subseteq EC$
3. Unique numbering condition:
   $\forall u, v, v' \in V$ and $e, e' \in (EC \cup ES)$ where $v \stackrel{\star}{\Longrightarrow} u$, $v' \stackrel{\star}{\Longrightarrow} u$, $e \neq e'$: If $\exists w, w', l, l'$ such that $e = (v \stackrel{l}{\longrightarrow} w)$ or $e = (v \rightarrow w)$, and $e' = (v' \stackrel{l'}{\longrightarrow} w')$ or $e' = (v' \rightarrow w')$, then $Ord(e) \neq Ord(e')$

When we refer to a class graph in the following we mean a legal class graph, unless we specifically mention illegality.

The cycle-free alternation condition is natural and has been proposed by other researchers, e.g., [31, page 396], [35, page 109: Class names may not depend on themselves in a circular fashion involving only (alternation) class productions]. The condition says that a class may not inherit from itself.

The unique labels condition guarantees that "inherited" construction edges are uniquely labeled and excludes class graphs which contain the patterns shown in Figure 15.

Other mechanisms for uniquely naming the construction edges could be used, e.g., the renaming mechanism of Eiffel and the overriding of part classes [26]. The theory does not seem to be affected significantly by small changes such as this.

The unique numbering condition is similar to the unique labels condition. It guarantees that the construction and syntax edges inherited at any vertex are totally ordered.

A.2. Object graphs

We have defined the concept of a class graph which mathematically captures some of the structural knowledge which object-oriented programmers use. Next we define object graphs and their relation to class graphs. An object graph defines a hierarchical object and is motivated by the interpretation of an object graph, called the standard interpretation, given in Table 2.

Definition. *An* **object graph**, *ψ, is a directed graph $\psi = (W, W_s, S, \Lambda_\psi; E, E_s, \lambda, Ord)$ where:*

- W is a finite set of vertices.
- W_s is a set of strings called the syntax vertices.
- S is an arbitrary finite set.
- Λ_ψ is a set of labels.
- E is a ternary relation on $W \times W \times \Lambda_\psi$. If $(v \stackrel{l}{\longrightarrow} w) \in E$ we call l the label of the edge $(v \stackrel{l}{\longrightarrow} w)$. No two edges outgoing from the same vertex may have the same label. That is, $\forall v, w, w' \in W, l \in \Lambda_\psi$ such that $w \neq w' : \{(v \stackrel{l}{\longrightarrow} w), (v \stackrel{l}{\longrightarrow} w')\} \not\subseteq E$
- E_s is a binary relation on $W \times W_s$ called the syntax edges.
- $\lambda : W \rightarrow S$ is a function that maps each vertex of ψ to an element of S.
- $Ord : (E \cup E_s) \rightarrow \mathcal{N}$ is a function that maps each edge to a natural number.

Normally, the set S is a subset of the construction vertices of some class graph. In the standard interpretation, the function λ maps each object in an object graph to the class

TABLE 2. Standard interpretation for object graphs

Graph	Object-oriented Design
vertex	object
immediate successor	immediate subpart or component
edge label	part name

of which it is an instance. We use a graphical notation for object graphs similar to that for class graphs. Vertices are represented by circles and edges by labeled arrows. The vertices are labeled with their class names (their mapping under λ). In case we wish to distinguish more than one instance of a class, the labels may be prefixed with an instance name followed by a ":".

Not every object graph with respect to a class graph is legal; intuitively, the object structure has to be consistent with the class definitions. Each object can only have parts as prescribed in the class definition and the parts prescribed in the class definitions must appear in the objects (see Figure 16).

Definition. *Let $p_1, p_2, \ldots, p_n$ be the outgoing edges (including syntax edges) from a vertex, $v \in W$, of an object graph such that Ord$(p_i) <$ Ord$(p_{i+1}), 1 \leq i < n$. Then the $\boldsymbol{PartOrder(v, p_i) = i}$.*

Let $q_1, q_2, \ldots, q_n$ be the construction and syntax edges outgoing from all vertices, v', from which a vertex, $v \in VC$ of a class graph is alternation reachable, such that Ord$(q_i) <$ Ord$(q_{i+1}), 1 \leq i < n$. Then the $\boldsymbol{PartOrder(v, q_i) = i}$.

Definition. *An object graph, $\psi = (W, W_s, S, \Lambda_\psi; E, E_s, \lambda,$ Ord$)$, is* **legal** *with respect to a class graph, $\phi = (VC, VA, VS, \Lambda; EC, EA, ES,$ Ord$)$, iff for each vertex, $v \in W$:*

- $\lambda(v) \in VC$
  Each vertex in the object graph maps to a construction vertex in the class graph.
- $\forall (r \xrightarrow{l} s) \in EC$ where $r \stackrel{\star}{\Longrightarrow} \lambda(v) : \exists w \in W$ such that $(v \xrightarrow{l} w) \in E$
  Each object has all of the sub-objects prescribed by the class graph.
- $\forall (r \to s) \in ES$ where $r \stackrel{\star}{\Longrightarrow} \lambda(v) : (v \to s) \in E_s$
  Each object has all of the concrete syntax prescribed by the class graph.

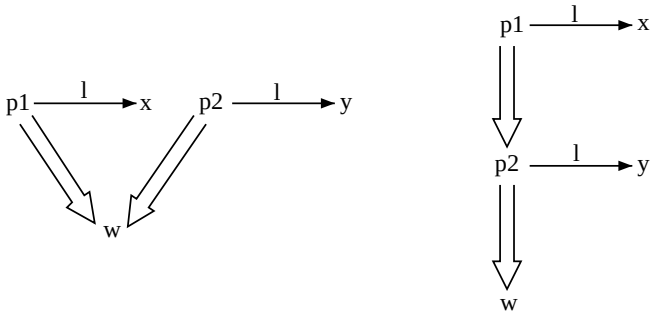


FIG. 15. Forbidden subgraphs

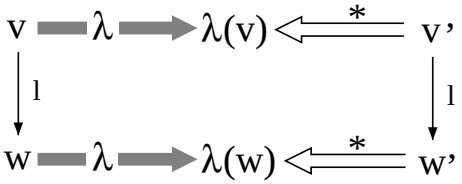


FIG. 16. Legality Rule

- $\forall w, l$ where $(v \xrightarrow{l} w) \in E : \exists (r \xrightarrow{l} s) \in EC$ such that $r \stackrel{\star}{\Longrightarrow} \lambda(v), s \stackrel{\star}{\Longrightarrow} \lambda(w)$, and $PartOrder(v, (v \xrightarrow{l} w)) = PartOrder(\lambda(v), (r \xrightarrow{l} s))$
  Each object has *only* the sub-objects prescribed by the class graph and has them in the proper order.
- $\forall s$ where $(v \to s) \in E_s : \exists r$ such that $(r \to s) \in ES, r \stackrel{\star}{\Longrightarrow} \lambda(v)$, and $PartOrder(v, (v \to w)) = PartOrder(\lambda(v), (r \to s))$
  Each object has *only* the concrete syntax prescribed by the class graph and has it in the proper order.

Example 2. Consider the graphs in Figure 17. The object graph, ψ, is legal with respect to the class graph, ϕ. The object graph is given by: $W = \{i1, i2, i3, i4\}, E = \{(i1 \xrightarrow{b} i2), (i1 \xrightarrow{c} i3), (i2 \xrightarrow{bc} i4)\}, \Lambda_\psi = \{b, bc, c\}, \lambda = \{i1 \to A, i2 \to B, i3 \to C, i4 \to C\}$.

Example 3. Consider object graphs in Figure 19 which are illegal with respect to the class graph in Figure 18. The first object graph is illegal since apples don't contain stones and the second because `Cherry` *is not alternation-reachable from* `Number`.

The language of a class graph, ϕ, is formally defined in terms of *sentences* which are defined, in turn, by the object graphs which are legal with respect to ϕ.

Definition. *An acyclic object graph, ψ, rooted at a unique vertex, v, has a textual representation, called $\boldsymbol{sentence(\psi)}$,*

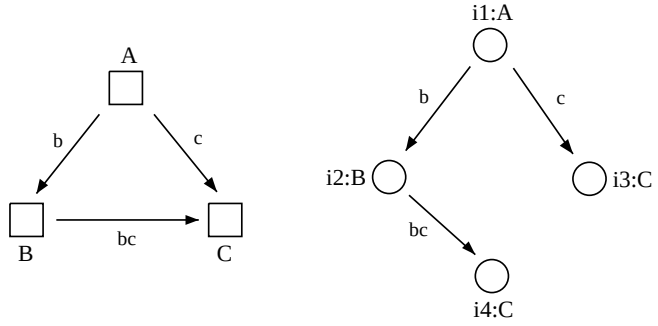


FIG. 17. Class graph, ϕ, and legal object graph, ψ.

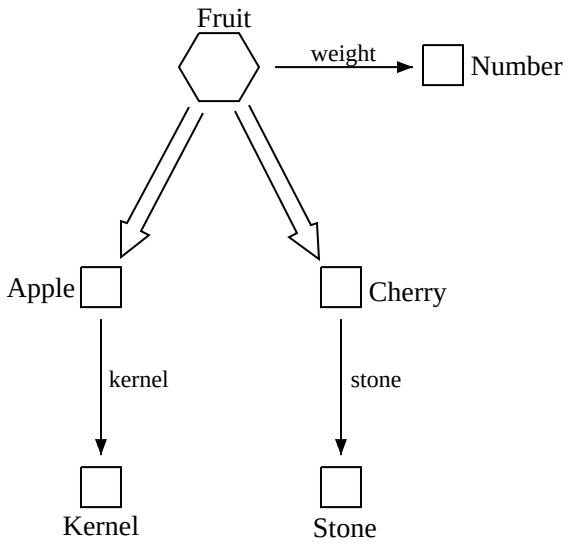


FIG. 18. Fruit class graph

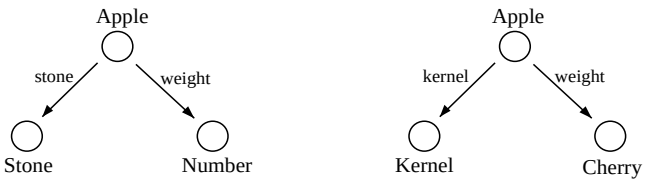


FIG. 19. Fruit object graphs

which is the string of syntax vertices encountered during a depth first traversal of ψ starting from v. If an object graph, ψ, is cyclic or unrooted, $sentence(\psi)$ is undefined.

We say that an object graph, $\psi = (W, W_s, S, \Lambda_\psi; E, E_s, \lambda, Ord)$, is **rooted** *at vertex v if v is the only vertex in W with an in-degree of 0.*

Definition. *The* **language** *defined by a class graph, $\phi = (V, VS, \Lambda; EC, EA, ES, Ord)$, is given by:*

$$L(\phi) = \{s | \exists \psi : s = sentence(\psi), \text{ and } \psi \text{ is legal with respect to } \phi\}$$

The **input language** *of a CG program, P, with class graph, ϕ, is given by:*

$$L(P) = \{s | \exists \psi : s = sentence(\psi), \psi \text{ is legal with respect to } \phi, \text{ and } \psi \text{ is rooted at } Main\}$$

Definition. *The set of all legal object graphs with respect to a class graph, ϕ, is called $\boldsymbol{Objects(\phi)}$.*

The definitions above relate a class graph with a set of object graphs. In object-oriented programming language terminology, a class graph corresponds to a set of class definitions and the object graphs correspond to the objects which can be created calling "constructor" functions of the classes. In some languages, e.g., C++, the class definitions considerably restrict the objects which can be created. The definitions above demand even more discipline than C++.

In the context of evolution, we often wish to discuss object graphs that are not legal with respect to the current class graph. We sometimes refer to these object graphs as *object example graphs* since our goal is often to modify a class graph so that it will become compatible with a new set of objects based on examples.

A.3. Related work

The axiomatic model which is used in this paper is new but similar data models exist in the literature. In particular, the notions of "alternation" and "construction" appear as "classification" and "aggregation" in both Hull and Yap's Format Model [16] and Kuper and Vardi's LDM [20]. Ait-Kaci's feature structures [2] are also related to the Demeter kernel model. Our abstraction algorithms [24, 6] can be adapted to abstract feature structures from examples.

Other related work in the data base field is described in: [1, 9, 37].

Appendix B

Formal Definition of Primitives

B.1. Object-preserving transformations

The definitions of the object-preserving transformations for class graphs are as follows:

- **Renumbering of parts**. Any set of construction and syntax edges in a class graph, ϕ, may be renumbered (by replacing the *Ord* function) to produce a new class graph, ϕ', if for all vertices, $v \in V$, and edges, $e \in (EC \cup ES)$, such that v is alternation reachable from the source of e: $PartOrder_\phi(v, e) = PartOrder_{\phi'}(v, e)$.
- **Abstraction of common parts**. If $\exists v, w, l, i$ such that $\forall v'$, where $(v \Longrightarrow v') \in EA : (v' \xrightarrow{l} w) = e \in EC \text{and} Ord(e) = i$, or $(v' \rightarrow w) = e \in ES \text{and} Ord(e) = i$, then all of the edges, e, can be replaced by a new edge, e', with v as its source and $Ord(e') = Ord(e)$.
  Intuitively, if all of the immediate subclasses of class C have the same part, that part can be moved up the inheritance hierarchy so that each of the subclasses will inherit the part from C, rather than duplicating the part in each subclass.
- **Distribution of common parts**. An outgoing construction edge, $e = (v \xrightarrow{l} w)$, or syntax edge, $e = (v \rightarrow w)$, can be deleted from an alternation vertex, v, if for each $(v \Longrightarrow v') \in EA$ a new construction edge $e' = (v' \xrightarrow{l} w)$, or syntax edge $e' = (v' \rightarrow w)$, respectively is added with $Ord(e') = Ord(e)$.
  This is the inverse of abstraction of common parts.
- **Deletion of "useless" alternation**. An alternation vertex is "useless" if it has no incoming edges and no outgoing construction edges. If an alternation vertex is useless it may be deleted along with its outgoing alternation edges.

Intuitively, an alternation vertex is useless if it is not a part of any construction class, and it has no parts for any construction class to inherit.

- **Addition of "useless" alternation**. An alternation vertex, v, can be added along with outgoing alternation edges to any set of vertices already in the class graph. This is the inverse of deletion of useless alternation.
- **Part replacement**. If the set of construction vertices which are alternation-reachable from some vertex, $v \in V$, is equal to the set of construction vertices alternation-reachable from another vertex, $v' \in V$, then any construction edge $(w \xrightarrow{l} v) \in EC$ can be deleted and replaced with a new construction edge, $(w \xrightarrow{l} v')$.
  Intuitively, if two class C1 and C2 have the same set of instantiable (construction) subclasses then the defined objects do not change when C1 is replaced by C2 in a part definition. Note that the inverse of part replacement is just another instance of the transformation.

B.2. Renaming of vertices and edges

Any construction edge, $(v \xrightarrow{l} w) \in EC$ may be replaced by a construction edge with a different label, $(v \xrightarrow{l'} w)$. Also, any construction vertex, $v \in VC$, may be replaced by a different construction vertex, v', with the same incoming and outgoing edges. When viewing a picture of a class graph it appears that the vertex has been "renamed" by changing its label. Since the labels or identifiers used to denote construction vertices have a global scope, and the same identifiers may be used to denote vertices in other class graphs, a changed label implies a changed vertex. On the other hand, since the identifiers used to denote alternation vertices have a scope local to the class graph, changing the labels of alternation vertices may be done freely, but does not in any way "transform" the class graph.

B.3. Nesting of parts

Given a vertex, $w \in V$ with no incoming alternation edges and a different vertex, $u \in (V \cup VS)$, such that for every construction edge, $e_v = (v \rightarrow w')$, where $w \stackrel{\star}{\Longrightarrow} w'$, there is a syntax or construction edge, $e'_v = (v \rightarrow u)$, and w' has at most one incoming alternation edge, then:

- If for each v, $PartOrder(v, e'_v) = PartOrder(v, e_v) + 1$, then we may delete each edge, e'_v, and add a single replacement edge, e, from w to u and let $Ord(e) > Ord(e')$ for all other construction and syntax edges, e' outgoing from any w' where $w \stackrel{\star}{\Longrightarrow} w'$.
  Intuitively, if every class which has w as a part has u as a part immediately after w, then we may remove the u part from all of those classes and instead make u the last part of class w. See, for example, Figure 3.
- If for each v, $PartOrder(v, e'_v) = PartOrder(v, e_v) - 1$, then we may delete each edge, e'_v, and add a single replacement edge, e, from w to u and let $Ord(e) < Ord(e')$ for all other construction and syntax edges, e' outgoing from any w' where $w \stackrel{\star}{\Longrightarrow} w'$.
  Intuitively, if every class which has w as a part has u as a part immediately before w, then we may remove the u part from all of those classes and instead make u the first part of class w.

B.4. Unnesting of parts

Given a vertex, $w \in V$ with no incoming alternation edges and an outgoing construction edge or syntax edge, e, with target u:

- If for every construction or syntax edge, $e' \neq e$, with source w' such that $w \stackrel{\star}{\Longrightarrow} w'$, w' has at most one incoming alternation edge and $Ord(e) > Ord(e')$ (so e is the last part of every w object), then we may delete edge e, if for each construction from some vertex, v, to w, e_v, we add a replacement edge, e'_v, from v to u such that $PartOrder(v, e'_v) = PartOrder(v, e_v) + 1$. In other words, we remove the last part, p, from every w object, and insert the part p just after the w part of every object that *contains* a w object.
  or
- If for every construction or syntax edge, $e' \neq e$, with source w' such that $w \stackrel{\star}{\Longrightarrow} w'$, w' has at most one incoming alternation edge and $Ord(e) < Ord(e')$ (so e is the first part of every w object), then we may delete edge e, if for each construction from some vertex, v, to w, e_v, we add a replacement edge, e'_v, from v to u such that $PartOrder(v, e'_v) = PartOrder(v, e_v) - 1$. In other words, we remove the first part, p, from every w object, and insert the part p just before the w part of every object that *contains* a w object.

This is the inverse of *nesting of parts*.

B.5. Addition of lambda parts

From any vertex, $v \in V$, an outgoing construction edge, $(v \xrightarrow{l} w)$ to a construction vertex, $w \in VC$, may be added if w has no outgoing edges. An outgoing syntax edge, $(v \rightarrow w)$ to a syntax vertex, w, may be added if w is the "empty string".

B.6. Deletion of lambda parts

A construction edge whose target is a construction vertex with no outgoing edges, or a syntax edge whose target is the "empty string" may be deleted. This is the inverse of *addition of lambda parts*.

B.7. Addition of lambda alternative

An alternation edge, $(v \Longrightarrow w)$, may be added from an alternation vertex, $v \in VA$ to a construction vertex, $w \in VC$ if w has no outgoing edges, v has only one outgoing construction edge, $(v \xrightarrow{l} v')$, and the target, v', of that edge has an outgoing alternation edge, $(v' \Longrightarrow v)$, back to v. See, for example, Figure 4.

B.8. Deletion of lambda alternative

An alternation edge, $(v \Longrightarrow w)$, from a vertex, $v \in$ *VA* to a construction vertex, $w \in$ *VC*, may be deleted if w has no outgoing edges, v has only one outgoing construction edge, $(v \xrightarrow{i} v')$, and the target, v', of that edge has an outgoing alternation edge, $(v' \Longrightarrow v)$, back to v. This is the inverse of *addition of lambda alternative*.

B.9. Insertion of singleton construction

A new construction vertex, v, with a single outgoing construction edge to a vertex, $v' \in V$, may be added to a class graph, and any incoming construction edges at v' may be rerouted to v. Incoming alternation edges at v' may also by rerouted to v if the rerouting does not result in the inheritance of additional parts (syntax or construction edges) at v. See, for example, Figure 5.

B.10. Deletion of singleton construction

If a class graph contains a construction vertex, v, with no inherited parts and a single outgoing edge to a vertex, $v' \in (V \cup$ *VS*), then v may be deleted if all incoming edges at v are rerouted to v'. This is the inverse of *insertion of singleton construction*.

B.11. Attribute to subclass

If a class graph contains a construction vertex, $v \in$ *VC*, with an outgoing construction edge, $(v \xrightarrow{i} w)$, to an alternation vertex, $w \in$ *VA*, then we may delete the construction edge from v to w and for each vertex, w', such that $(w \Longrightarrow w') \in$ *EA*, we add a new construction vertex, v', with an incoming alternation edge from v, $(v \Longrightarrow v')$, and an outgoing construction edge, $(v' \xrightarrow{i} w')$ to w'. Each of the new construction edges is mapped to the same number (under *Ord*) as was the deleted construction edge. Since v now has outgoing alternation edges it becomes (by definition) an alternation vertex. See, for example, Figure 6.

B.12. Subclass to attribute

If a class graph contains alternation vertices, $v, w \in$ *VA*, such that there is a one to one correspondence between the vertices, v', where $(v \Longrightarrow v') \in$ *EA* and the vertices, w' where $(w \Longrightarrow w') \in$ *EA*, such that for each w' the corresponding v' is a construction vertex with a single incoming edge, $(v \Longrightarrow v')$, and a single outgoing edge, $(v' \xrightarrow{i} w')$, then we may delete each such v' along with its incoming and outgoing edges and add a new construction edge, $(v \xrightarrow{i} w)$, from v to w. Since v no longer has any outgoing alternation edges it becomes (by definition) a construction vertex. This is the inverse of *attribute to subclass*.

Appendix C

Primitive transformations for completeness proof

Figures 20 - 28 show sequences of primitive transformations that correspond to each of the equations in the axiom system, $\mathcal{F}$, except for the equations $(\alpha \cdot \phi) = \phi$ and $(\alpha + \phi) = \alpha$ which are not applicable since the regular class graphs as defined here do not contain the empty language, ϕ.

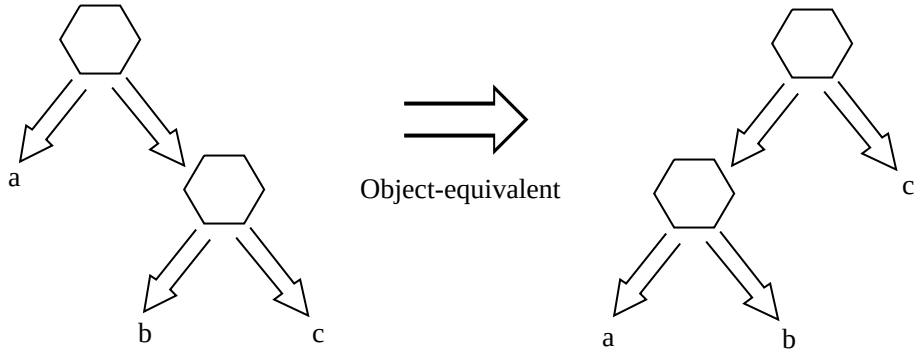


FIG. 20. $(a + (b + c)) = ((a + b) + c)$

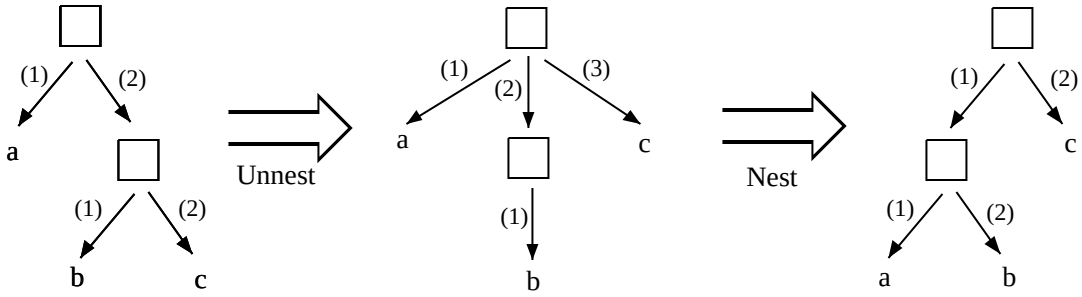


FIG. 21. $(a \cdot (b \cdot c)) = ((a \cdot b) \cdot c)$

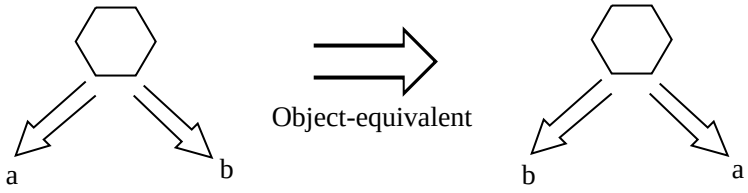


FIG. 22. $(a + b) = (b + a)$

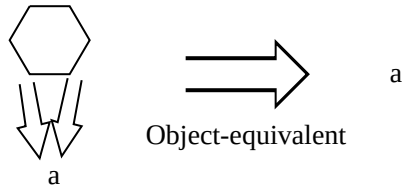
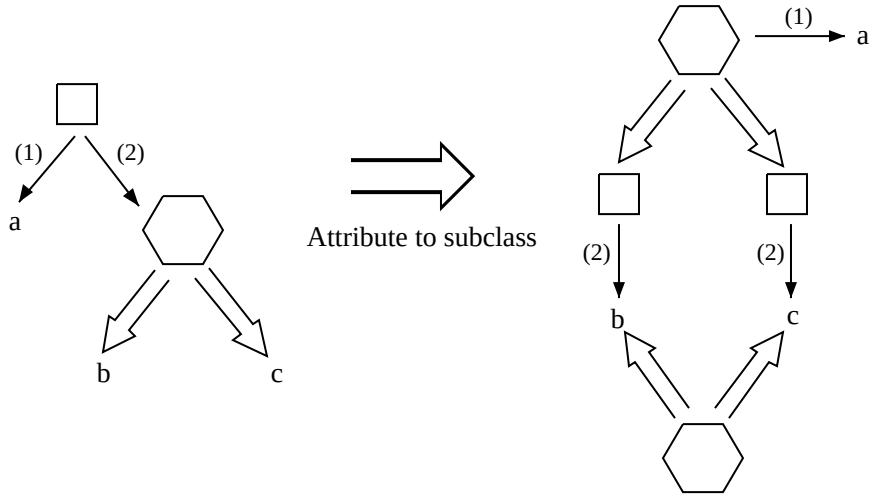


FIG. 23. $(a + a) = a$



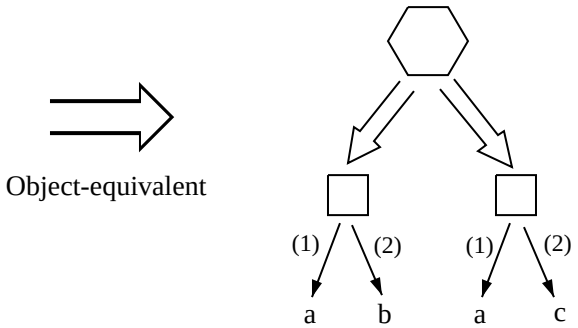


FIG. 24. $(a \cdot (b + c)) = ((a \cdot b) + (a \cdot c))$

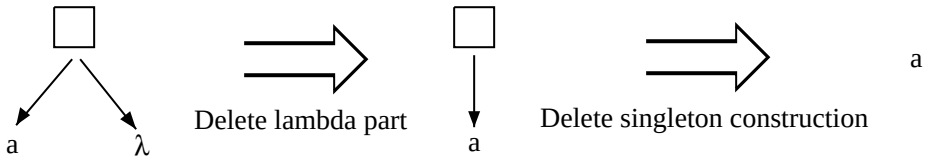
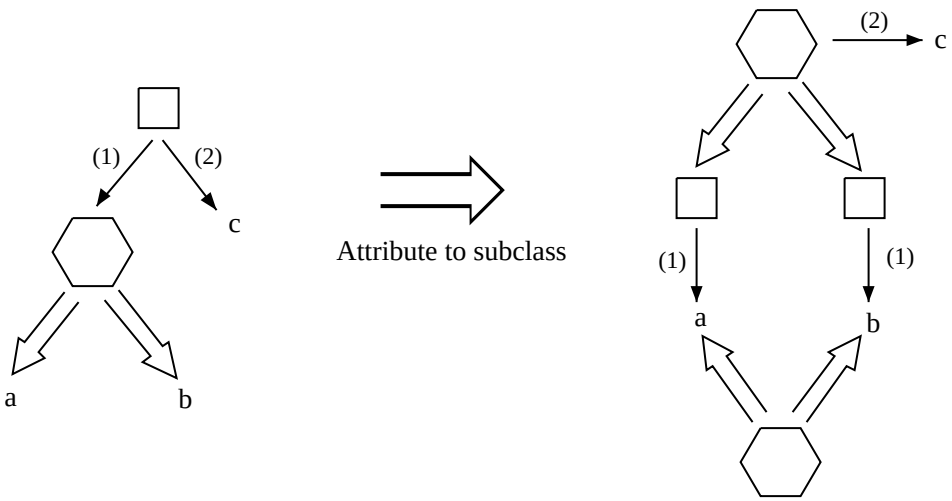


FIG. 25. $(a \cdot \lambda) = a$



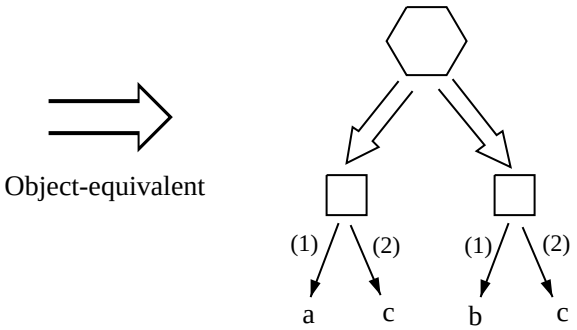


FIG. 26. $((a + b) \cdot c) = ((a \cdot c) + (b \cdot c))$

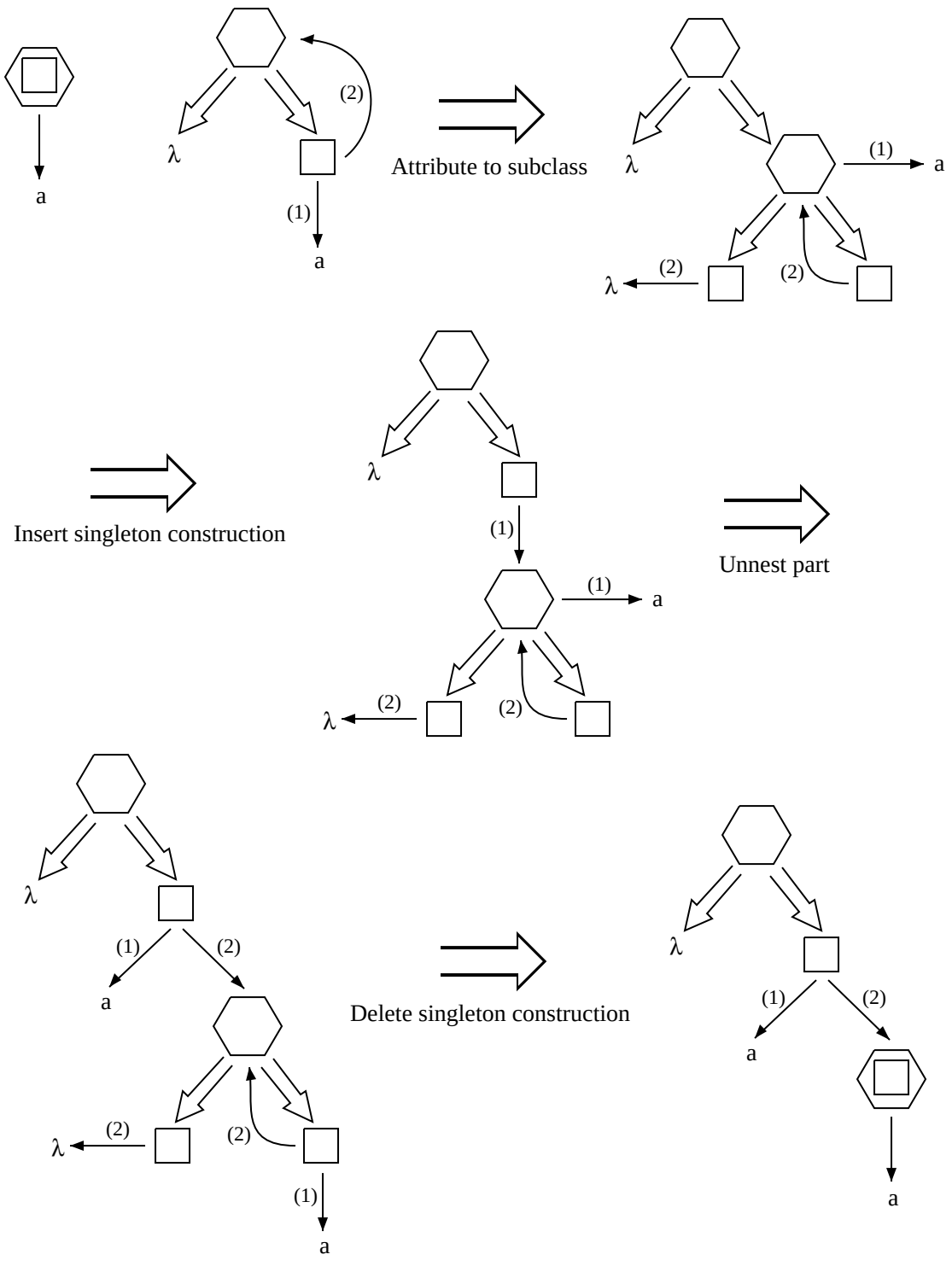


FIG. 27. $(a*) = (\lambda + (a \cdot (a*)))$

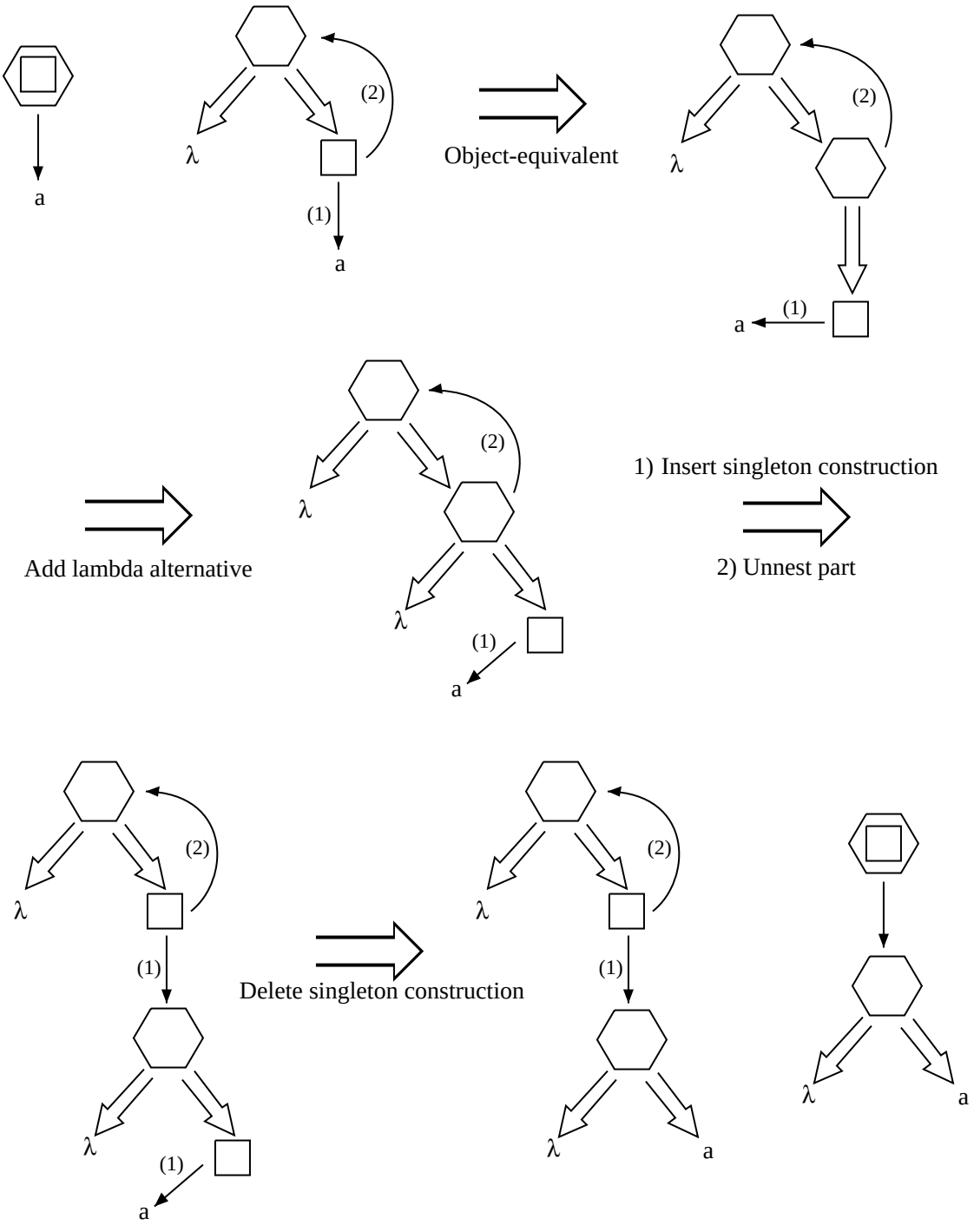


FIG. 28. $(a*) = ((\lambda + a)*)$