Reengineering Class Hierarchies Using Concept Analysis

Gregor Snelting
Technische Universität Braunschweig
Abteilung Softwaretechnologie
Bültenweg 88, D-38106 Braunschweig, Germany
snelting@ips.cs.tu-bs.de

Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
tip@watson.ibm.com

Abstract

The design of a class hierarchy may be imperfect. For example, a class C may contain a member m not accessed in any C-instance, an indication that m could be eliminated, or moved into a derived class. Furthermore, different subsets of C's members may be accessed from different C-instances, indicating that it might be appropriate to split C into multiple classes. We present a framework for detecting and remediating such design problems, which is based on *concept analysis*. Our method analyzes a class hierarchy along with a set of applications that use it, and constructs a *lattice* that provides valuable insights into the usage of the class hierarchy in a specific context. We show how a restructured class hierarchy can be generated from the lattice, and how the lattice can serve as a formal basis for interactive tools for redesigning and restructuring class hierarchies.

1 Introduction

Designing a class hierarchy is hard, because it is not always possible to anticipate how a hierarchy will be used by an application. This is especially the case when a class hierarchy is developed as a library, and designed independently from the applications that use it. Ongoing maintenance, in particular ad-hoc extensions of the hierarchy, will further increase the system's entropy. As typical examples of inconsistencies that may arise, one might think of:

- A class C may contain a member m not accessed in any C-instance, an indication that m may be removed, or moved into a derived class.
- Different instances of a given class C may access different subsets of C's members, an indication that it might be appropriate to split C into multiple classes.

In this paper, we present a method for analyzing the usage of a class hierarchy based on *concept analysis* [28]. Our approach comprises the following steps. First, a table is constructed that precisely reflects the usage of a class hierarchy. In particular, the table makes explicit relationships between the types of variables, and class members such as "the type of x must be a base class of the type of y", and "member m must occur in a base class of the type of variable x" are encoded in the table. From the table, a *concept lattice* is derived, which factors out information that variables or members have in common. We will show how the concept lattice can provide valuable insight into the design of the class hierarchy, and how it can serve as a basis for automated or interactive restructuring tools for class hierarchies. The examples presented in this paper are written in C++, but our approach is applicable to other object-oriented languages as well.

Our method can analyze a class hierarchy along with any number of programs that use it, and provide the user with either a combined view reflecting the usage of the hierarchy by the entire set of programs, or with individual views that clarify how each application uses the hierarchy. Analyzing a class hierarchy without any accompanying applications is also possible, and can be useful to study the internal dependences inside class definitions.

1.1 A motivating example

Consider the example of Figure 1, which is concerned with relationships between students and professors. Figure 1(a) shows a class hierarchy, in which a class `Person` is defined that contains a person's `name`, `address`, and `socialSecurityNumber`. Classes `Student` and `Professor` are derived from `Person`. `Students` have an identification number (`studentId`), and a thesis `advisor` if they are graduate students. A constructor is provided for initializing `Students`, and a method `setAdvisor` for designating a `Professor` as an advisor. `Professors` have a `faculty` and a `workAddress`, and a professor may hire a student as a teaching `assistant`. A constructor is provided for initialization, and a method `hireAssistant` for hiring a `Student` as an assistant. Details for classes `Address` and `String` are not provided; in the subsequent analysis these classes will be treated as "atomic" types and we will not attempt to analyze them.

Figure 1(b) and (c) show two programs that use the class hierarchy of Figure 1(a). In the first program, a student and a professor are created, and the professor is made the student's advisor. The second program creates another student and professor, and here the student is made the professor's assistant. The example is certainly not perfect C++ code, but looks reasonable enough at first glance.

Figure 2 shows the lattice computed by our method for the class hierarchy and the two example programs of Figure 1. Ignoring a number of details, the lattice may be interpreted as follows:

- The lattice elements (concepts) may be viewed as *classes* of a restructured class hierarchy that precisely reflects the usage of the original class hierarchy by the client programs.
- The ordering between lattice elements may be viewed as *inheritance* relationships in the restructured class hierarchy.

To appear in the Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, FL, November 3-5, 1998.

```
class String { /* details omitted */ };
class Address { /* details omitted */ };
enum Faculty { Mathematics, ComputerScience };
class Professor; /* forward declaration */

class Person {
public:
  String name;
  Address address;
  long socialSecurityNumber;
};

class Student :  public Person {
public:
  Student(String sn, Address sa, int si){
    name = sn; address = sa; studentId = si;
  };
  void setAdvisor(Professor *p){
    advisor = p;
  };
  long studentId;
  Professor *advisor;
};
class Professor :  public Person {
public:
  Professor(String n, Faculty f, Address wa){
    name = n; faculty = f;
    workAddress = wa;
    assistant = 0; /* default:  no assistant */
  };
  void hireAssistant (Student *s){
    assistant = s;
  };
  Faculty faculty;
  Address workAddress;
  Student *assistant; /* either 0 or 1 assistants */
};
```

(a)

```
int main(){
  String s1name, p1name;
  Address s1addr, p1addr;
  Student* s1 =                        /* Student1 */
    new Student(s1name,s1addr,12345678);
  Professor *p1 =                      /* Professor1 */
    new Professor(p1name,Mathematics,p1addr);
  s1->setAdvisor(p1);
  return 0;
}
```

(b)

```
int main(){
  String s2name, p2name;
  Address s2addr, p2addr;
  Student* s2 =                        /* Student2 */
    new Student(s2name,s2addr,87654321);
  Professor *p2 =                      /* Professor2 */
    new Professor(p2name, ComputerScience, p2addr);
  p2->hireAssistant(s2);
  return 0;
}
```

(c)

Figure 1: Example: relationships between students and professors. **(a)** Class hierarchy for expressing associations between students and professors. **(b)** Example program using the class hierarchy of Figure 1(a). **(c)** Another example program using the class hierarchy of Figure 1(a).

- A variable v has type C in the restructured class hierarchy if v occurs immediately *below* concept C in the lattice.
- A member m occurs in class C if m appears *directly above* concept C in the lattice.

Examining the lattice of Figure 2 according to this interpretation reveals the following interesting facts[1]:

- Data member `Person::socialSecurityNumber` is never accessed, because no variable appears below it. This illustrates situations where subclassing is used to inherit the functionality of a class, but where some of that functionality is not used.
- Data member `Person::address` is only used by students, and not by professors (for professors, the data member `Professor::workAddress` is used instead, perhaps because their home address is confidential information). This illustrates a situation where the member of a base class is used in some, but not all derived classes.
- No members are accessed from parameters `s` and `p`, and from data members `advisor` and `assistant`. This is due to the fact that no operations are performed on a student's advisor, or on a professor's assistant. Such situations are typical of redundant, incomplete, or erroneous code and should be examined closely.
- The analyzed programs create professors who hire assistants (`Professor2`), and professors who do not hire assistants (`Professor1`). This can be seen from the fact that method `Professor::hireAssistant()` appears above the concept labeled `Professor2`, but not above the concept labeled `Professor1`.
- There are students with advisors (`Student1`) and students without advisors (`Student2`). This can be seen from the fact that data member `Student::setAdvisor` appears above the concept labeled `Student1`, but not above the concept labeled `Student2`.
- Class `Student`'s constructor does not initialize the `advisor` data member. This can be seen from the fact that attribute `Student::advisor` does not appear above attribute `Student::Student()` in the lattice[2].

One can easily imagine how the above information might be used as the basis for restructuring the class hierarchy. One possibility would be for a tool to automatically generate restructured source code from the information provided by the lattice, similar to the approach taken in [26, 27]. However, from a redesign perspective, we believe that an interactive approach would be more appropriate. For example, the programmer doing the restructuring job may decide that the data member `socialSecurityNumber` should be retained in the class hierarchy because it may be needed later. In the interactive tool we envision, one could indicate this by *moving up* in the lattice the attribute under consideration, `socialSecurityNumber`. The programmer may also decide that certain fine distinctions in the lattice are unnecessary. For example, one may decide that it is not necessary to distinguish between professors that hire assistants, and professors that don't. In an interactive tool, this distinction could be removed by *merging* the concepts for `Professor1` and `Professor2`.

[1] The labels `Student1`, `Professor1`, `Student2`, and `Professor2` that appear in the lattice represent the types of the heap objects created by the example programs at various program points (indicated in Figures 1(b) and (c) using comments).

[2] `Student::Student()` also represents the `this`-pointer of the method.

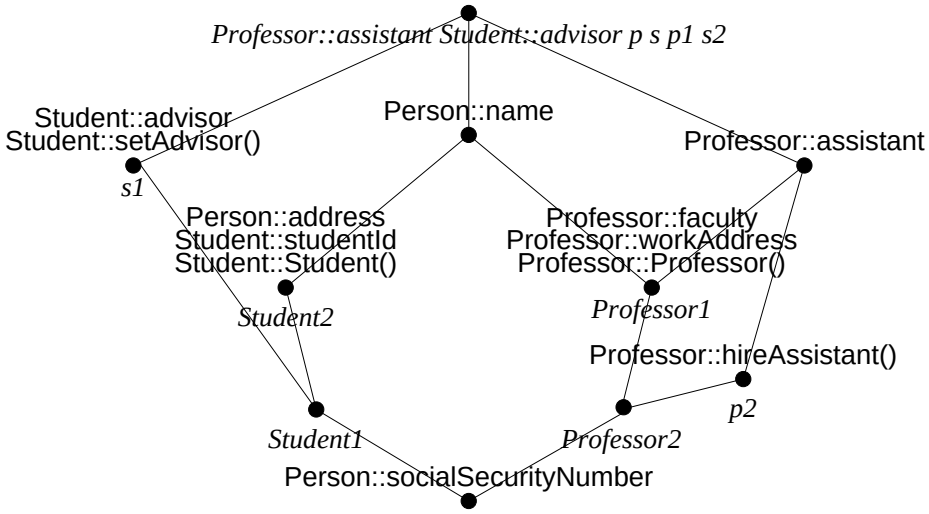

Figure 2: Lattice for Student/Professor example.

Another useful capability of an interactive tool would be to associate names with lattice elements. When the programmer is done manipulating the lattice, these names could be used as class names in the restructured hierarchy when the restructured source code is generated. For example, using the information provided by the lattice, the programmer may determine that `Student` objects on which the `setAdvisor` method is invoked are graduate students, whereas `Student` objects on which this method is not called are undergraduates. Consequently, he may decide to associate names `Student` and `GraduateStudent` with the concepts labeled `Student2` and `Student1`, respectively.

1.2 Organization of this paper

The remainder of this paper is organized as follows. Section 2 briefly reviews the relevant parts of the theory of concept analysis. In Section 3 we define the objects and attributes in our domain, which correspond to the rows and columns of the tables. The process of constructing tables and lattices is presented in Section 4. In Section 5, we discuss how the information provided by the lattice can reveal problems in the design of class hierarchies, and how the lattice can be used as a basis for interactive tools for restructuring class hierarchies. Section 6 discusses related work. Finally, conclusions and directions for future work are presented in Section 7.

2 Concept Analysis

Concept analysis provides a way to identify groupings of *objects* that have common *attributes*. The mathematical foundation was laid by Birkhoff in 1940 [3]. Birkhoff proved that for every binary relation between certain objects and attributes, a lattice can be constructed that provides remarkable insight into the structure of the original relation. The lattice can always be transformed back to the original relation, hence concept analysis is similar in spirit to Fourier analysis. Later, Wille and Ganter elaborated Birkhoff's result and transformed it into a data analysis method [28, 6]. Since then, it has found a variety of applications, including analysis of software structures [9, 21, 11, 20, 7].

Concept analysis starts with a relation, or boolean table, T between a set of *objects* $\mathcal{O}$ and a set of *attributes* $\mathcal{A}$, hence $T \subseteq \mathcal{O} \times \mathcal{A}$. For any set of objects $O \subseteq \mathcal{O}$, their set of common attributes is defined as $\sigma(O) = \{a \in \mathcal{A} \mid \forall o \in O : (o, a) \in T\}$. For any set of attributes $A \subseteq \mathcal{A}$, their set of common objects is $\tau(A) = \{o \in \mathcal{O} \mid \forall a \in A : (o, a) \in T\}$.

A pair (O, A) is called a *concept* if $A = \sigma(O)$ and $O = \tau(A)$. Informally, such a concept corresponds to a *maximal rectangle* in the table T: any $o \in O$ has all attributes in A, and all attributes $a \in A$ fit to all objects in O. It is important to note that concepts are invariant against row or column permutations in the table. The set of all concepts of a given table forms a partial order via $(O_1, A_1) \leq (O_2, A_2) \iff O_1 \subseteq O_2 \iff A_1 \supseteq A_2$. Birkhoff proved that the set of concepts constitutes a complete lattice, the *concept lattice* $\mathcal{L}(T)$. For two elements (O_1, A_1) and (O_2, A_2) in the concept lattice, their infimum or *meet* is defined as

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

and their supremum or *join* as

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

A concept $c = (O, A)$ has *extent* $\mathit{ext}(c) = O$ and *intent* $\mathit{int}(c) = A$. In our figures, a lattice element (concept) c is labeled with attribute $a \in \mathcal{A}$, if it is the *largest* concept with a in its intent, and it is labeled with an object $o \in \mathcal{O}$, if it is the *smallest* concept with o in its extent. The (unique) lattice element labeled with a is denoted $\mu(a) = \bigvee\{c \in \mathcal{L}(T) \mid a \in \mathit{int}(c)\}$, and the (unique) lattice element labeled with o is denoted $\gamma(o) = \bigwedge\{c \in \mathcal{L}(T) \mid o \in \mathit{ext}(c)\}$. The following property establishes the connection between a table and its lattice, and shows that they can be reconstructed from each other:

$$(o, a) \in T \iff \gamma(o) \leq \mu(a)$$

Hence, the attributes of object o are those which appear *above* o, and all objects that appear *below* a have attribute a. Consequently, join points (suprema) in the lattice indicate that certain objects have attributes in common, while meet points (infima) show that certain attributes fit to common objects. In other words, join points factor out common attributes, while meet points factor out common objects. Thus, the lattice uncovers a hierarchy of conceptional clusters that was implicit in the original table.

Figure 3 shows a table and its lattice (taken from [5]). The element labeled *far* corresponds to the maximal rectangle indicated in the table. This element is the supremum of all elements with *far* in their intent: *Pluto*, *Jupiter*, *Saturn*, *Uranus*, *Neptune* are below

	small	medium	large	near	far	moon	no moon
Mercury	×			×			×
Venus	×			×			×
Earth	×			×		×	
Mars	×			×		×	
Jupiter			×		×	×	
Saturn			×		×	×	
Uranus		×			×	×	
Neptune		×			×	×	
Pluto	×				×	×	

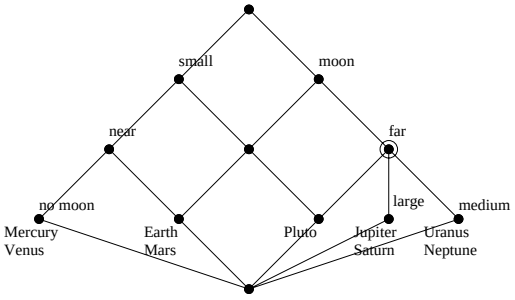

Figure 3: Example table and associated concept lattice.

far in the lattice, and the table confirms that these (and no other) planets are indeed far away.

A table and its lattice are alternate views on the same information, serving different purposes and providing different insights. There is yet another view: a set of *implications*. Let $A, B \subseteq \mathcal{A}$ be two sets of attributes. We say that *A implies B*, iff any object with the attributes in A also has the attributes in B:

$$A \to B \iff \forall o \in \mathcal{O} : (\forall a \in A : (o, a) \in T) \Rightarrow (\forall b \in B : (o, b) \in T)$$

For $B = \{b_1, \ldots, b_k\}$, $A \to B$ holds iff $A \to b_i$ for all $b_i \in B$.[3] Implications show up in the lattice as follows: $A \to b$ holds iff $\bigwedge\{\mu(a) \mid a \in A\} \leq \mu(b)$. Informally, implications between attributes can be found along upward paths in the lattice. In the example of Figure 3, we have that $\mu(\text{far}) \leq \mu(\text{moon})$, which can be read as *far* $\to$ *moon*, or "A planet which is far away has a moon". Other examples of implication are *no moon* $\to$ *near*, *small*; or *near*, *far*$\to$ *large* (the latter implication being true because its premise is contradictory).

Often, some implications are known to hold *a priori*. Such background knowledge can easily be integrated into a given table. An implication $x \to y$ can be enforced by adding the entries in the x column to the y column, and will cause $\mu(x) \leq \mu(y)$ in $\mathcal{L}(T)$. A general implication $A \to B$ can be enforced by adding the intersection of the A columns to all B columns.

Implications between objects can be defined (and enforced) similarly. For any lattice, a *minimal implication base* can be computed, which allows to obtain all the other implications by applying propositional logic.

Construction of concept lattices and implication bases has typical time complexity $O(n^3)$ for an $n \times n$ table, but can be exponential in the worst case. Empirical studies show that even for large tables, exponential behavior is extremely rare [21]. In fact it can be shown that if the number of attributes for every object is bounded (which is true for most applications), the lattice size is linear in the number of table entries.

There is much more to say about concept lattices, their structure theory, and related algorithms and methodology. Davey and Priestley's book [5] contains a chapter on elementary concept analysis. Ganter and Wille [6] treat the topic in depth. Section 4.7 will sketch Ganter's algorithm for lattice construction.

3 Objects and Attributes

Roughly speaking, the objects and attributes in our domain are variables and class members, respectively, and the table that will be constructed in Section 4 identifies for each variable which members must be included in its type. Before we can define the objects and attributes more precisely, we need to introduce some terminology. In what follows, $\mathcal{P}$ denotes a program containing a class hierarchy, or a collection of programs that share a class hierarchy. Further, v, w, ... denote the variables in $\mathcal{P}$ whose type is a class, and p, q, ... the variables in $\mathcal{P}$ whose type is a pointer to a class (references can be treated similarly, and will be ignored in the present paper). Expressions are denoted by x, y, ... We will henceforth use "variables" to refer to variables as well as parameters. In the definitions that follow, *TypeOf*$(\mathcal{P}, x)$ denotes the type of expression x in $\mathcal{P}$.

The *objects* of our domain are the program variables through which the class hierarchy is accessed. Variables whose type is (pointer to) built-in can be ignored because the class hierarchy can only be accessed through variables whose type is *class-related* (i.e., variables whose type is a class, or a pointer to a class). Definition 1 below defines sets of variables *ClassVars* and *ClassPtrVars* whose type is a class, and a pointer to a class, respectively. In Section 4.8, we will discuss how to model heap-allocated objects. Note that *ClassPtrVars* includes implicitly declared `this` pointers of methods. In order to distinguish between `this` pointers of different methods, we will henceforth refer to the `this` pointer of method `A::f()` by the fully qualified name of its method, i.e., `A::f`.

Definition 1 *Let $\mathcal{P}$ be a program. Then, we define the set of class-typed variables and the set of pointer-to-class-typed variables as follows:*

$$\textit{ClassVars}(\mathcal{P}) \triangleq \{\, v \mid v \text{ is a variable in } \mathcal{P},\ \textit{TypeOf}(\mathcal{P}, v) = C, \text{ for some class } C \text{ in } \mathcal{P} \,\}$$

$$\textit{ClassPtrVars}(\mathcal{P}) \triangleq \{\, p \mid p \text{ is a variable in } \mathcal{P},\ \textit{TypeOf}(\mathcal{P}, *p) = C, \text{ for some class } C \text{ in } \mathcal{P} \,\}$$

The *attributes* of our domain are class members. Following the definitions of [26, 27], we will distinguish between *definitions* and *declarations* of members. We define these terms as follows: The definition of a member comprises a member's signature (interface) as well as the executable code in its body, whereas the declaration of a member only represents its signature. This distinction is needed for accurately modeling virtual method calls. Consider a call to a virtual method f from a *pointer* p. In this case, only the declaration of f needs to be contained in p's type in order to be able to invoke f; the body of f does not need to be statically visible to p[4]. Naturally, a *definition* of f must be visible to the object that p

[3] We will usually write $a_1, \ldots, a_n \to b_1, \ldots, b_m$ instead of $\{a_1, \ldots, a_n\} \to \{b_1, \ldots, b_m\}$.

[4] Our objective is to identify the smallest possible set of member declarations and definitions that must be included in the type of any variable. Including the *definition* of f in $*p$'s type may lead to the incorporation of members that are otherwise not needed (in particular, members accessed from f's `this` pointer).

```
class A {
public:
  virtual int f(){ return g(); };
  virtual int g(){ return x; };
  int x;
};
class B : public A {
public:
  virtual int g(){ return y; };
  int y;
};
class C : public B {
public:
  virtual int f(){ return g() + z; };
  int z;
};

int main(){
  A a; B b; C c;
  A *ap;
  if (...)  { ap = &a; }
  else { if (...)  { ap = &b; }
         else { ap = &c; } }
  ap->f();
  return 0;
}
```

Figure 4: Example program $\mathcal{P}_1$.

points to at run-time, so that the dynamic dispatch can be executed correctly.

Definition 2 (shown below) defines sets *MemberDcls*($\mathcal{P}$) and *MemberDefs*($\mathcal{P}$) of member declarations and member definitions in $\mathcal{P}$. We distinguish between declarations and definitions of virtual methods for the reasons stated above. For nonvirtual methods, making this distinction is not necessary because the full definition of a nonvirtual method must always be statically visible to the caller. Therefore, nonvirtual methods are modeled using definitions only. Data members are modeled as declarations because they have no `this` pointer from which other members can be accessed.

Definition 2 *Let $\mathcal{P}$ be a program. Then, we define the set of member declarations and member definitions as follows:*

$$\textit{MemberDcls}(\mathcal{P}) \triangleq \{\ \textit{dcl}(C::m) \mid \textit{m is a data member or virtual method in class } C\ \}$$

$$\textit{MemberDefs}(\mathcal{P}) \triangleq \{\ \textit{def}(C::m) \mid \textit{m is a virtual or nonvirtual method in class } C\ \}$$

Example: Figure 4 shows a program $\mathcal{P}_1$ that will be used as a running example. For $\mathcal{P}_1$, we have:

$$\begin{aligned}
\textit{ClassVars}(\mathcal{P}_1) &\equiv \{\ \texttt{a},\ \texttt{b},\ \texttt{c}\ \}\\
\textit{ClassPtrVars}(\mathcal{P}_1) &\equiv \{\ \texttt{ap},\ \texttt{A::f},\ \texttt{A::g},\ \texttt{B::g},\ \texttt{C::f}\ \}\\
\textit{MemberDcls}(\mathcal{P}_1) &\equiv \{\ \textit{dcl}(\texttt{A::f}),\ \textit{dcl}(\texttt{A::g}),\ \textit{dcl}(\texttt{A::x}),\\
&\qquad \textit{dcl}(\texttt{B::g}),\ \textit{dcl}(\texttt{B::y}),\ \textit{dcl}(\texttt{C::f}),\\
&\qquad \textit{dcl}(\texttt{C::z})\ \}\\
\textit{MemberDefs}(\mathcal{P}_1) &\equiv \{\ \textit{def}(\texttt{A::f}),\ \textit{def}(\texttt{A::g}),\ \textit{def}(\texttt{B::g}),\\
&\qquad \textit{def}(\texttt{C::f})\ \}
\end{aligned}$$

In Section 4.9, we will discuss how class-typed data members (which behave like variables because other members can be accessed from them) are modeled.

4 Table and Lattice Construction

This section describes how tables and lattices are constructed. Recall that the purpose of the table is to record for each variable the set of members that are used. A few auxiliary definitions will presented first, in Section 4.1.

4.1 Auxiliary definitions

For each variable v in *ClassPtrVars*($\mathcal{P}$) we will need a conservative approximation of the variables in *ClassVars*($\mathcal{P}$) variables that v may point to. Any of several existing algorithms [4, 16, 23, 19]) can be used to compute this information, and we do not make assumptions about the particular algorithm used to compute points-to information. Definition 3 expresses the information supplied by some points-to analysis algorithm as a set *PointsTo*($\mathcal{P}$), which contains a pair $\langle p, v\rangle$ for each pointer p that may point to a class-typed variable v.

Definition 3 *Let $\mathcal{P}$ be a program. Then, the points-to information for $\mathcal{P}$ is defined as follows:*

$$\textit{PointsTo}(\mathcal{P}) \triangleq \{\ \langle p, v\rangle \mid p \in \textit{ClassPtrVars}(\mathcal{P}),\ v \in \textit{ClassVars}(\mathcal{P}),\ p \textit{ may point to } v\ \}$$

Example: We will use the following points-to information for program $\mathcal{P}_1$. Recall that $X::f$ denotes the `this` pointer of method $X::f()$.

$$\begin{aligned}
\textit{PointsTo}(\mathcal{P}_1) \equiv \{\ &\langle \texttt{ap},\texttt{a}\rangle,\ \langle \texttt{ap},\texttt{b}\rangle,\ \langle \texttt{ap},\texttt{c}\rangle,\ \langle \texttt{A::f},\texttt{a}\rangle,\ \langle \texttt{A::f},\texttt{b}\rangle,\\
&\langle \texttt{C::f},\texttt{c}\rangle,\ \langle \texttt{A::g},\texttt{a}\rangle,\ \langle \texttt{B::g},\texttt{b}\rangle,\ \langle \texttt{B::g},\texttt{c}\rangle\ \}
\end{aligned}$$

Note that the following simple algorithm suffices to compute the information of Example 4.1: for each pointer p of type $*X$, assume that it may point to any object of type Y, such that (i) $Y = X$ or Y is a class transitively derived from X, and (ii) if p is the `this` pointer of a virtual method $C::m$, no overriding definitions of m are visible in class Y.

We will use the following terminology for function and method calls. A *direct* call is any call to a function or a nonvirtual method, or an invocation of a virtual method from a variable in *ClassVars*($\mathcal{P}$). An *indirect* call is an invocation of a virtual method from a variable in *ClassPtrVars*($\mathcal{P}$) (requiring a dynamic dispatch).

4.2 Table entries for member access operations

Table T has a *row* for each element of *ClassVars*($\mathcal{P}$) and *ClassPtrVars*($\mathcal{P}$), and a *column* for each element of *MemberDcls*($\mathcal{P}$) and *MemberDefs*($\mathcal{P}$). Informally, an entry $(y, \textit{dcl}(A::m))$ appears in T iff the declaration of m is contained in y's type, and an entry $(y, \textit{def}(A::m))$ appears in T iff the definition of m is contained in y's type. We begin by adding entries to T that reflect the member access operations in the program. Definition 4 below defines a set *MemberAccess*($\mathcal{P}$) of all pairs $\langle m, y\rangle$ such that member m is accessed from variable y. For an *indirect* call $p \rightarrow f(y_1, \ldots, y_n)$, we also include an element $\langle f, y\rangle$ in *MemberAccess*($\mathcal{P}$) for each $\langle p, y\rangle \in \textit{PointsTo}(\mathcal{P})$.

Definition 4 *Let $\mathcal{P}$ be a program. Then, the set of member access operations in $\mathcal{P}$ is defined as follows:*

$$\begin{aligned}
\textit{MemberAccess}(\mathcal{P}) \triangleq\ &\{\ \langle m, v\rangle \mid v.m \textit{ occurs in } \mathcal{P},\ m \textit{ is a class member in } \mathcal{P},\\
&\qquad v \in \textit{ClassVars}(\mathcal{P})\ \} \cup\\
&\{\ \langle m, *p\rangle \mid p \rightarrow m \textit{ occurs in } \mathcal{P},\ m \textit{ is a class member in } \mathcal{P},\\
&\qquad p \in \textit{ClassPtrVars}(\mathcal{P})\ \} \cup\\
&\{\ \langle m, y\rangle \mid p \rightarrow m \textit{ occurs in } \mathcal{P},\ \langle p, y\rangle \in \textit{PointsTo}(\mathcal{P}),\\
&\qquad m \textit{ is a virtual method in } \mathcal{P}\ \}
\end{aligned}$$

Example: For program $\mathcal{P}_1$ of Figure 4, we have:

$$\begin{array}{l} \mathit{MemberAccess}(\mathcal{P}_1) \equiv \\ \{\ \langle \texttt{x}, \texttt{*A::g}\rangle,\ \langle \texttt{y}, \texttt{*B::g}\rangle,\ \langle \texttt{z}, \texttt{*C::f}\rangle,\ \langle \texttt{g}, \texttt{*A::f}\rangle,\ \langle \texttt{g}, \texttt{*C::f}\rangle, \\ \quad \langle \texttt{f}, \texttt{*ap}\rangle,\ \langle \texttt{f}, \texttt{a}\rangle,\ \langle \texttt{f}, \texttt{b}\rangle,\ \langle \texttt{f}, \texttt{c}\rangle,\ \langle \texttt{g}, \texttt{a}\rangle,\ \langle \texttt{g}, \texttt{b}\rangle,\ \langle \texttt{g}, \texttt{c}\rangle\ \} \end{array}$$

Accessing a class member is not an entirely trivial operation because different classes in a class hierarchy may contain members with the same name (or signature). Furthermore, in the presence of multiple inheritance, an object may contain multiple subobjects of a given type C, and hence multiple members $C::m$. This implies that whenever a member m is accessed, one needs to determine which m is being selected. This selection process is defined informally in the C++ Draft Standard [1] as a set of rules that determine when a member *hides* or *dominates* another member with the same name. Rossie and Friedman [18] provided a formalization of the member lookup, as a function on *subobject graphs*. This framework has subsequently been used by Tip et al. as a formal basis for operations on class hierarchies such as slicing [25] and specialization [26]. Ramalingam and Srinivasan recently presented an efficient algorithm for member lookup [17].

For the purposes of the present paper, we will assume the availability of a function *static-lookup* which, given a class C and a member m, determines the base class B (B is either C, or a transitive base class of C) in which the selected member is located[5]. For details on function *static-lookup*, the reader is referred to [18, 25].

We are now in a position to state how the appropriate relations between variables and declarations and definitions should be added to the table:

Definition 5 *Let $\mathcal{P}$ be a program with associated table T. Then, the following entries are added to the table due to member access operations that occur in the program.*

$$\frac{\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{DataMembers}(\mathcal{P}),\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)}{(y, \mathit{dcl}(X::m)) \in T}$$

$$\frac{\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{NonVirtualMethods}(\mathcal{P}),\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)}{(y, \mathit{def}(X::m)) \in T}$$

$$\frac{\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{VirtualMethods}(\mathcal{P}),\ y \equiv *p,\ p \in \mathit{ClassPtrVars}(\mathcal{P}),\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)}{(y, \mathit{dcl}(X::m)) \in T}$$

$$\frac{\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{VirtualMethods}(\mathcal{P}),\ y \equiv v,\ v \in \mathit{ClassVars}(\mathcal{P}),\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)}{(y, \mathit{def}(X::m)) \in T}$$

4.3 Table entries for `this` pointers

The next table construction rule we will present is concerned with `this` pointers of methods. Consider the fact that for each method $C::f()$, there is a column in the table labeled $\mathit{def}(C::f)$, and a row labeled $*C::f$. The former is used to express the fact that method $C::f()$ may be called from objects. The latter is necessary to reflect members being accessed from method $C::f()$'s `this` pointer. Unless precautions are taken, the attribute $\mathit{def}(C::f)$ and the object $*C::f$ may appear at different points in the lattice, though

[5] In [18, 25], *static-lookup* is defined as a function from subobject to subobjects. Since the present paper is only concerned with the *classes* in which members are located, we will simply ignore all subobject information below.

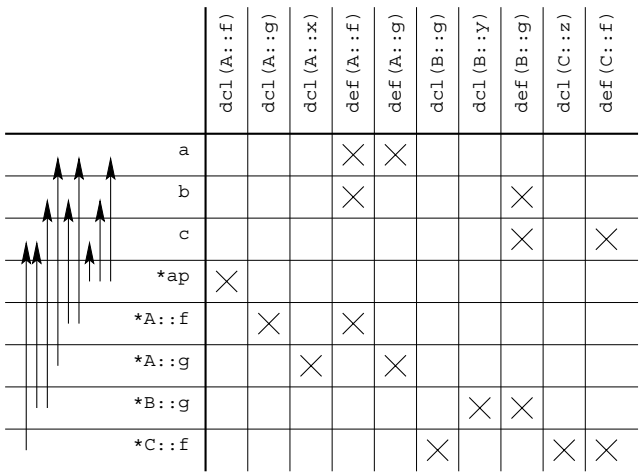

Table 1: Initial table for program $\mathcal{P}_1$ of Figure 4. Arrows indicate implications due to assignments (see Section 4.4).

$\gamma(*C::f) \geq \mu(\mathit{def}(C::f))$ must always hold[6]. In such cases, our method effectively infers that the type of a `this` pointer could be a base class of the type in which method $C::f$ occurs (and therefore be less constrained). However, in reality, the type of a method's `this` pointer is *determined by* the class in which the associated method definition appears.

The table entries added by Definition 6 will force a method's attribute and a method's `this` pointer to appear at the same lattice element; by ensuring $\gamma(*C::f) \leq \mu(\mathit{def}(C::f))$. This will allow us later to remove rows for `this` pointers from the table when constructing the lattice.

Definition 6 *Let $\mathcal{P}$ be a program. Then, the following entries are added to the table:*

$$\frac{\mathit{def}(C::m) \in \mathit{MemberDefs}(\mathcal{P})}{(*C::m, \mathit{def}(C::m)) \in T}$$

Example: Table 1 shows the table for program $\mathcal{P}_1$ of Figure 4 after adding the entries according to Definitions 5 and 6.

4.4 Table entries for assignments

Consider an assignment $x = y$, where $x \equiv v$ and $y \equiv w$, for some class-typed variables $v,\ w \in \mathit{ClassVars}(\mathcal{P})$. Such an assignment is only valid if the type of x is a base class of the type of y. Consequently, any member declaration or definition that occurs in x's type must also occur in y's type. We will enforce this constraint using an *implication* from the row for x to the row for y. However, we will begin by formalizing the notion of an assignment.

Definition 7 below defines a set $\mathit{Assignments}(\mathcal{P})$ that contains a pair of objects $\langle v, w\rangle$ for each assignment $v = w$ in $\mathcal{P}$ where v and w are class-typed. In addition, $\mathit{Assignments}(\mathcal{P})$ also contains entries for cases where the type of the left-hand side and/or the right-hand side of the assignment are a pointer to a class. Parameter-passing in direct calls to functions and methods is modeled by way of assignments between corresponding formal and actual parameters. For an *indirect* call $p \rightarrow f(y_1, \ldots, y_n)$, $\mathit{Assignments}(\mathcal{P})$ contains additional elements that model the parameter-passing in the direct call $x.f(y_1, \ldots, y_n)$, for each $\langle p, x\rangle \in \mathit{PointsTo}(\mathcal{P})$. That is, we conservatively approximate the potential targets of dynamically dispatched calls. The set $\mathit{Assignments}(\mathcal{P})$ will also contain

[6] See Appendix.

elements for implicit parameters such as `this` pointers of methods and function/method return values whose type is class-related.

Definition 7 *Let $\mathcal{P}$ be a program. Then, the set of assignments between variables whose type is a (pointer to a) class is defined as follows:*

$$
\begin{aligned}
&\mathit{Assignments}(\mathcal{P}) \triangleq \\
&\quad \{\, \langle v, w\rangle \mid v = w \text{ occurs in } \mathcal{P},\ v,\ w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, w\rangle \mid\ p = \&w \text{ occurs in } \mathcal{P},\ p \in \mathit{ClassPtrVars}(\mathcal{P}), \\
&\qquad\qquad w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, *q\rangle \mid p = q \text{ occurs in } \mathcal{P},\ p,\ q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, w\rangle \mid\ *p = w \text{ occurs in } \mathcal{P},\ p \in \mathit{ClassPtrVars}(\mathcal{P}), \\
&\qquad\qquad w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle v, *q\rangle \mid\ v = *q \text{ occurs in } \mathcal{P},\ v \in \mathit{ClassVars}(\mathcal{P}), \\
&\qquad\qquad q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, *q\rangle \mid *p = *q \text{ occurs in } \mathcal{P},\ p,\ q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\}
\end{aligned}
$$

Example: For program $\mathcal{P}_1$ of Figure 4, we have:

$$
\begin{aligned}
&\mathit{Assignments}(\mathcal{P}_1) \equiv \\
&\quad \{\ \langle \texttt{*ap}, \texttt{a}\rangle,\ \langle \texttt{*ap}, \texttt{b}\rangle,\ \langle \texttt{*ap}, \texttt{c}\rangle,\ \langle \texttt{*A::f}, \texttt{a}\rangle,\ \langle \texttt{*A::f}, \texttt{b}\rangle, \\
&\qquad \langle \texttt{*C::f}, \texttt{c}\rangle,\ \langle \texttt{*A::g}, \texttt{a}\rangle,\ \langle \texttt{*B::g}, \texttt{b}\rangle,\ \langle \texttt{*B::g}, \texttt{c}\rangle \,\}
\end{aligned}
$$

We are now in a position to express how elements should be added to the table due to assignments. Definition 8 states this as an *implication*, which tells us how elements should be copied from one row to another.

Definition 8 *Let $\mathcal{P}$ be a program with associated table T. Then, the following implications must be encoded in the table due to assignments that occur in $\mathcal{P}$:*

$$\frac{\langle x, y\rangle \in \mathit{Assignments}(\mathcal{P})}{x \to y}$$

Example: For program $\mathcal{P}_1$ of Figure 4, the following assignment implications are generated:

`*ap` $\to$ `a`, `*ap` $\to$ `b`, `*ap` $\to$ `c`, `*A::f` $\to$ `a`, `*A::f` $\to$ `b`,
`*C::f` $\to$ `c`, `*A::g` $\to$ `a`, `*B::g` $\to$ `b`, `*B::g` $\to$ `c`

These implications are indicated on the left side of Table 1. Table 2 is obtained by copying the elements from the "source row" to the "target row" according to each of these implications.

4.5 Table entries for preserving dominance/hiding

The table thus far encodes for each variable the members contained in its type (either directly because a member is accessed from that variable, or indirectly due to assignments between variables). However, in the original class hierarchy, an object's type may contain more than one member with a given name. In such cases, the member lookup rules of [1] determine which member is accessed. This is expressed as a set of rules that determine when a member *hides* or *dominates* another member with the same name. In cases where a variable contains two members m that have a hiding relationship in the original class hierarchy, this hiding relationship must be preserved, because we are interested in generating a restructured hierarchy from the table, and the member access operations in the program might otherwise become ambiguous. Definition 9 incorporates the appropriate hiding/dominance relations into the table, using implications between attributes:

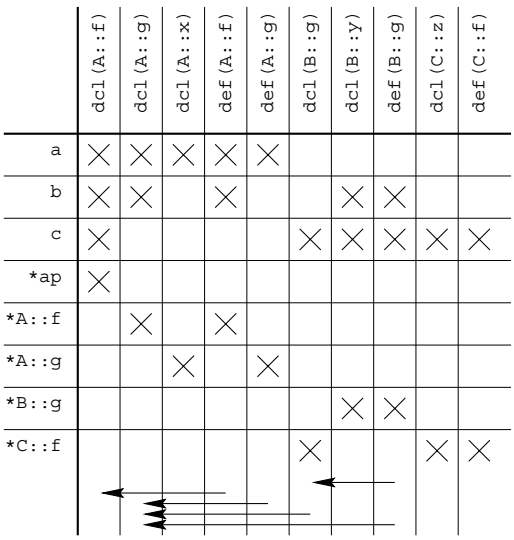

Table 2: Table after application of assignment implications. Arrows indicate implications for preserving hiding/dominance among members with the same name (see Section 4.5).

Definition 9 *Let $\mathcal{P}$ be a program with associated table T. Then, the following implications are incorporated into T in order to preserve hiding and dominance:*

$$\frac{(x, \mathit{dcl}(A{::}m)) \in T,\ (x, \mathit{dcl}(B{::}m)) \in T,\ A \text{ is a transitive base class of } B}{\mathit{dcl}(B{::}m) \to \mathit{dcl}(A{::}m)}$$

$$\frac{(x, \mathit{dcl}(A{::}m)) \in T,\ (x, \mathit{def}(B{::}m)) \in T,\ A = B \text{ or } A \text{ is a transitive base class of } B}{\mathit{def}(B{::}m) \to \mathit{dcl}(A{::}m)}$$

$$\frac{(x, \mathit{def}(A{::}m)) \in T,\ (x, \mathit{def}(B{::}m)) \in T,\ A \text{ is a transitive base class of } B}{\mathit{def}(B{::}m) \to \mathit{def}(A{::}m)}$$

$$\frac{(x, \mathit{def}(A{::}m)) \in T,\ (x, \mathit{dcl}(B{::}m)) \in T,\ A \text{ is a transitive base class of } B}{\mathit{dcl}(B{::}m) \to \mathit{def}(A{::}m)}$$

Example: For program $\mathcal{P}_1$, the following dominance implications are generated:

$$
\begin{array}{ll}
\mathit{def}(\texttt{A::f}) \to \mathit{dcl}(\texttt{A::f}) & \mathit{def}(\texttt{A::g}) \to \mathit{dcl}(\texttt{A::g}) \\
\mathit{def}(\texttt{B::g}) \to \mathit{dcl}(\texttt{A::g}) & \mathit{dcl}(\texttt{B::g}) \to \mathit{dcl}(\texttt{A::g}) \\
\mathit{def}(\texttt{B::g}) \to \mathit{dcl}(\texttt{B::g}) &
\end{array}
$$

These implications are shown at the bottom of Table 2. After incorporating these implications, Table 3 results.

Remark: Observe that the implication $\mathit{def}(\texttt{B::g}) \to \mathit{dcl}(\texttt{A::g})$ only becomes necessary after propagating table elements according to the other implications.

4.6 Efficiently applying implications

Since the assignment implications can generate new dominance implications and vice versa, it seems that a fixpoint iteration is necessary in order to compute the final table. Fortunately, there is a direct algorithm for table completion which runs in time $O(|\mathcal{O}| \times |\mathcal{A}|)$.

	dcl(A::f)	dcl(A::g)	dcl(A::x)	def(A::f)	def(A::g)	dcl(B::g)	dcl(B::y)	def(B::g)	dcl(C::z)	def(C::f)
a	×	×	×	×	×					
b	×	×		×		×	×	×		
c	×	×				×	×	×	×	×
*ap	×									
*A::f	×	×		×						
*A::g		×	×		×					
*B::g		×				×	×	×		
*C::f		×				×			×	×

Table 3: Final table for program $\mathcal{P}_1$.

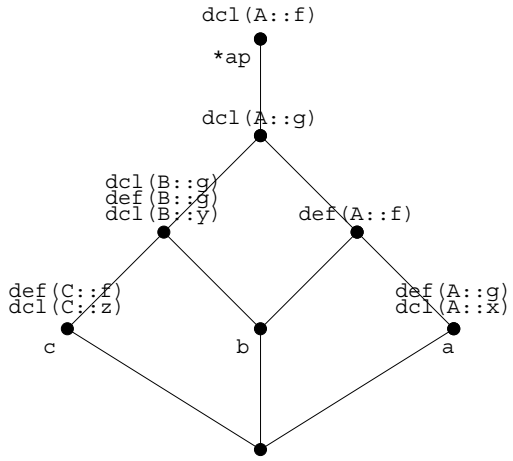

Figure 5: Lattice for program $\mathcal{P}_1$, generated from Table 3 after removing the rows labeled *A::f, *A::g, *B::g, and *C::f.

The algorithm is based on the fact that any set of dependencies (implications) between rows or columns can be lifted to dependencies between individual table entries: an implication $o_1 \rightarrow o_2$ generates table entry dependencies $(o_1, a) \rightarrow (o_2, a)$ for all $(o_1, a) \in T$; similarly, $a_1 \rightarrow a_2$ generates table entry dependencies $(o, a_1) \rightarrow (o, a_2)$ for all $(o, a_1) \in T$. Table entries are processed in topological order according to the lifted dependencies. Since only positive entries are propagated, cycles can be ignored: a backward edge closing a dependency cycle must lead to a table entry which has been set earlier (otherwise the cycle would not have been explored, due to the topological ordering).

4.7 Lattice construction

From the final table, the lattice can be constructed using Ganter's algorithm [6]. This algorithm utilizes the fact that both $\sigma \circ \tau$ and $\tau \circ \sigma$ are closure operators. Ganter's algorithm computes the closed sets for any closure operator by enumerating all subsets of $\mathcal{O}$ (or $\mathcal{A}$) in alphabetical order[7]. The closure operator is applied to every subset, and whenever the result of the closure operator changes, a new closed subset of $\mathcal{O}$ has been found. Enumeration then continues, beginning with the new closed subset. This process enumerates all closed subsets—that is, concept extents—in alphabetical order. By applying σ the intents are obtained, and finally all concepts are arranged in a partial order via mutual comparison of their extents, and their labels are computed. In practice, Ganter's algorithm needs less than a second for 2000-element lattices on a standard workstation [21]. Subsequent layout of the lattice graph is much more expensive.

There is one minor issue that deserves mentioning. Recall that in Section 4.3 table entries were added to ensure that method definitions and their `this` pointers show up at the same lattice element. In order to avoid presenting redundant information to the user, we will henceforth omit `this` pointers from the lattice. The easiest way to accomplish this is to remove the rows for `this` pointer variables to the table prior to generating the lattice. Note that rows for `this` pointers cannot be left out during table construction because they are needed to model member accesses from `this` pointers, and the elements in such rows may be involved in implications due to assignments and dominance.

Example: Figure 5 shows the lattice for program $\mathcal{P}_1$, generated from Table 3 after removing the rows labeled *A::f, *A::g, *B::g, and *C::f. It demonstrates that a does not access B::y and C::z, while b and c do not access A::x and b does not access C::z. Similarly, the lattice shows the fine-grained differences in method access.

[7] This requires that the elements of $\mathcal{O}$ are ordered themselves somehow; subsets are then represented as strings of objects.

4.8 Heap allocation

Heap allocated objects can be handled in a straightforward way. Since our analysis is a static one, we are unable to distinguish between different objects created at the same allocation site. Our approach consists of simply treating each allocation site in the program as a class-typed variable (e.g., an element of the set *ClassVars*). For the program of Figure 1, there are four such allocation sites, which we refer to as `Student1`, `Student2`, `Professor1`, and `Professor2`.

4.9 Modeling nested objects

We conclude this section with a brief discussion of the treatment of *class-related data members* (i.e., data members whose type is class-related), such as `Student::advisor` in Figure 1. Like data members of built-in types, class-related data members can be accessed from variables and are therefore modeled as attributes. However, since other members may be accessed from a class-related data member, such data members play an alternate role as objects.

In order to clarify the issues involved in the reengineering of such "nested" structures, consider a class C that contains a data member m whose type is some class D. Then, the following information about m is made explicit in the concept lattice:

- The set of variables in which m is contained. This is modeled by treating m as an "attribute" a. Any object that occurs below a in the lattice contains m.
- The set of members contained in the type of m. This is modeled by treating the type of m as an "object" o. The set of members contained in o correspond to the attributes that occur above o in the lattice. This set of members is a subset of the members of D in the original class hierarchy.

Note that the "attribute view" of m corresponds exactly to the way we previously modeled data members with a built-in type, whereas the "object view" of m corresponds exactly to the way we previously modeled variables. The definitions that are concerned with

	dcl(Person::name)	dcl(Person:address)	dcl(Person::socialsecurityNumber)	dcl(Student::studentId)	dcl(Student::advisor)	dcl(Professor::faculty)	dcl(Professor::workAdress)	dcl(Profesor::assistant)	def(Student::Student)	def(Student::setAdvisor)	def(Professor::Professor)	def(Professor::hireAssistant)
*s1					×					×		
*s2												
*p1												
*p2								×				×
*s												
*p												
Student1	×	×		×	×				×	×		
Student2	×	×		×					×			
Professor1	×					×	×	×			×	
Professor2	×					×	×	×			×	×
*advisor												
*assistant												
*Student::Student	×	×		×					×			
*Student::setAdvisor					×					×		
*Professor::Professor	×					×	×	×			×	
*Professor::hireAssistant								×				×

Table 4: Final table for the Student/Professor example.

variables therefore apply to class-related data members as well, and for convenience we will henceforth assume the term "variable" to include class-related data members.

4.10 Modeling constructors

Constructors require special attention. A constructor generally initializes all data members contained in an object. If no constructor is provided by the user, a so-called default constructor is generated by the compiler, which performs the necessary initializations. The compiler may also generate a *call* to a constructor in certain cases. Modeling these compiler-generated actions as member access operations would lead us to believe that each member m of class C is needed in all C-instances, even in cases where the only access to m consists of its (default) initialization. Compiler-generated constructors, compiler-generated initializations, and compiler-generated calls to constructors will therefore be excluded from the set of member access operations. Destructors can be handled similarly.

4.11 Example

Table 4 shows the final table for the example of Figure 1, as obtained by analyzing the class hierarchy along with the two example programs. The lattice corresponding to this table was shown previously in Figure 2 (note that we replaced member definitions by the corresponding method names there for convenience).

4.12 Limitations

We conclude this section with a remark on a limitation of our analysis. In situations where an object x contains multiple subobjects of some type C (due to the use of nonvirtual multiple inheritance), our tables do not make a distinction between the various "copies" of the members of C in x. This leads to problems if the objective is to generate a new hierarchy from the lattice in which the distinct copies of the members of C must be preserved. We consider this to be a minor problem because situations where nonvirtual inheritance is used for its "member replicating" effect are quite rare in practice, and the restructuring tool could inform the user of the cases where the problem occurs. A clean solution to this problem would involve the encoding of subobject information in the table using an adaptation of the approach of [26, 27].

5 Restructuring class hierarchies

The following can be learned from the lattice (we refer the reader to the lattice of Figure 2 for examples):

- Data members that are not accessed anywhere in the program (e.g., `Person::socialSecurityNumber`) appear at the bottom element of the lattice.
- Data members of a base class B that are not used by (instances of) all derived classes of B are revealed. Such data members (e.g., `Person::address`) appear above (variables of) some but not all derived classes of B. For example, `Person::address` appears above instances of `Student`, but not above any instances of `Professor`.
- Variables from which no members are accessed appear at the the top element of the lattice (e.g., `s`).
- Data members that are properly initialized appear above the (constructor) method that is supposed to initialize them. If this is not the case, the data member may not be initialized. For example, we know that `Student::Student` does not initialize `Student::advisor` because that data member does not appear above `Student::Student` in the lattice.
- Situations where instances of a given type C access different subsets of C's members are revealed by the fact that variables of type C appear at different points in the lattice. Our example contains two examples of this phenomenon. The instances `Professor1` and `Professor2` of type `Professor` and the instances `Student1` and `Student2` of type `Student`.

The structure theory of concept lattices offers several algorithms which may provide useful information [22] as well. For example, one might think of measuring quality factors such as cohesion and coupling by algebraic decomposition of the lattice [11, 15].

As we mentioned earlier, a class hierarchy may be analyzed along with any number of programs, or without any program at all. The latter case may provide insights into the "internal structure" of a class library. Figure 6 shows the lattice obtained by analyzing the class hierarchy of Figure 1(a) *without* the programs of Figure 1(b) and (c); only code in method bodies is analyzed. Clearly, the resulting lattice should not be interpreted as a restructuring proposal, because it does not reflect the *usage* of the class hierarchy. However, there are some interesting things to note. For example, `socialSecurityNumber` is not accessed anywhere. If we would know in addition that `socialSecurityNumber` is private (i.e., that it can only be accessed by methods within its class), we could inform the user that it is effectively dead. Observe also that no members are accessed from method parameters `s` and

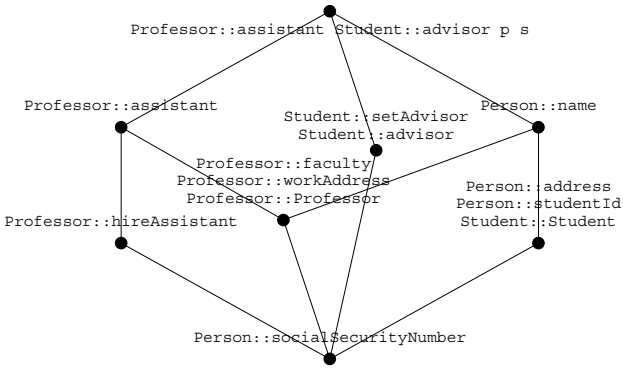

Figure 6: Lattice obtained by analyzing the class hierarchy of Figure 1 without accompanying programs.

p. Since the scope of these variables is local to the library, we know that analyzing additional code will not change this situation.

We intend to construct an interactive tool that provides the user with a view of the lattice and the associated table. One could easily imagine that such a tool would notify the user of anomalies in the design of a class hierarchy, as was discussed above. In addition, the tool could generate source-code from the lattice at any point in the transformation process by interpreting the lattice as a class hierarchy. Optionally, the analyzed programs could be transformed as well to take the new, restructured hierarchy into account. Specific transformations that the tool could support are:

- Unlabeled lattice elements (e.g., the center element in Figure 3) correspond to classes without members and without variables using them. The lattice can be simplified by pruning all such elements, and directly connecting their subordinate and superordinate neighbors. The fact that the resulting structure is not a lattice anymore, but only a partial order, is not relevant—lookup behavior and subobjects are not affected.
- The user can decide to merge[8] adjacent lattice elements if the distinction between these concepts is irrelevant (possibly because the lattice reflects a specific use of the hierarchy). For example, one may decide that the distinction between between professors that hire assistants, and professors that don't hire assistants is irrelevant, and therefore merge the concepts for `Professor1` and `Professor2`.
- With certain limitations, the user may move attributes upwards in the lattice, and object downwards. For example, the user may decide that `socialSecurityNumber` should be retained in the restructured class hierarchy, and move the corresponding attribute up to the concept labeled with attribute `Person::name`.
- Background knowledge that is not reflected in the lattice, e.g., "the type of x must be a base class of the type of y", can be integrated via background implications.
- Color should be used to display relevant substructures in the lattice, e.g., display all variables which formerly had the same type.
- The user may associate names with lattice elements. When the programmer is done manipulating the lattice, these names could be used as class names in the restructured hierarchy. For example, by examining the lattice, the programmer may determine that `Student` objects on which the `setAdvisor` method is invoked are graduate students, whereas `Student` objects on which this method is not called are undergraduates. Consequently, he may decide to associate names `Student` and `GraduateStudent` with the concepts labeled `s2` and `s1`, respectively.
- For very large class hierarchies, the tool could allow the user to focus on a selected subhierarchy either by specifying its minimal and maximal elements in the lattice, or by leaving out rows and columns in the table (in particular, the user could investigate the usage of a specific class C in the original hierarchy by focusing on the rows for the variables of type C, and the columns for the members of C).
- Very large lattices can also be subject to algebraic decomposition, such as horizontal decomposition, interference analysis or block relations [22]. Such decompositions correspond to natural subsystems of the original class hierarchy.

[8] There are some issues that a tool must take into account, because we want to preserve member lookup behavior. For example, merging two concepts that have different definitions of a virtual method f associated with them is not possible, because at most one f can occur in any class.

6 Related Work

6.1 Applications of concept analysis

Godin and Mili [7, 8] also use concept analysis for class hierarchy (re)design. The starting point in their approach is a set of interfaces of (collection) classes. A table is constructed that specifies for each interface the set of supported methods. The lattice derived from this table suggests how the design of a class hierarchy implementing these interfaces could be organized in a way that optimizes the distribution of methods over the hierarchy. Another property of their approach is that it identifies useful abstract classes that could be interesting in their own right, or suitable starting points for future extensions of the hierarchy. Although Godin and Mili's work has the same formal basis as ours, the domains under consideration are different. In [7], relations between members and classes are studied in order to improve the distribution of these members over the class hierarchy. In contrast, we study how the members of a class hierarchy are *used* in the executable code of a set of applications by examining relationships between variables and class members, and relationships among class members. Godin and Mili discuss some extensions of their basic approach to so-called multi-faceted domains, but do not study the usage of class hierarchies in applications.

Another application of concept analysis in the domain of software engineering is the analysis of software configurations. Snelting [21] uses concept analysis to analyze systems in which the C preprocessor (CPP) is used for configuration management. The relation between code pieces and governing expressions is extracted from a source file, and the corresponding lattice visualizes interferences between configurations. Later, Lindig proved that the configuration space itself is isomorphic to the lattice of the *inverted* relation [10].

Concept analysis was also used for modularization of old software. Siff and Reps [20] investigated the relation between procedures and "features" such as usage of global variables or types. A modularization is achieved by finding elements in the lattice whose intent partitions the feature space. Lindig and Snelting [11] also analyzed the relation between procedures and global variables in legacy Fortran programs. They showed that the presence of module candidates corresponds to certain decomposition properties of the lattice (the Siff/Reps criterion being a special case).

6.2 Class hierarchy specialization and application extraction

The work in the present paper is closely related to the work on *class hierarchy specialization* by Tip and Sweeney [26, 27]. Class hierarchy specialization is a space optimization technique in which a class hierarchy and a client program are transformed in such a way that the client's space requirements are reduced at run-time. The method of [26, 27] shares some basic "information gathering" steps with the method of the present paper[9], but the subsequent steps of that method are quite different. After determining the member access and assignment operations in the program, a set of *type constraints* is computed that capture the subtype-relationships between variables and members that must be retained. These type constraints roughly correspond to the information encoded in our tables, but contrary to our current approach they correctly distinguish between multiple subobjects that have the same type. From the type constraints, a new class hierarchy is generated automatically. In a separate step, the resulting class hierarchy is simplified by repeatedly applying a set of simple transformation rules.

In addition to the differences in the underlying algorithms, the method of [26, 27] differs from our reengineering framework in a number of ways. Class hierarchy specialization is an optimization technique that does not require any intervention by the user. In contrast, the current paper presents an *interactive* approach for analyzing the usage of a class hierarchy in order to find design problems. Reducing object size through the elimination of members is possible, but not necessarily an objective. For the purpose of restructuring it may very well be the case that an unused member should be retained in the restructured class hierarchy. The framework we presented here also allows for the analysis of a class hierarchy along with any number of programs, including none. Class hierarchy specialization customizes a class hierarchy w.r.t. a *single* client application.

Several other *application extraction* techniques for eliminating unused components from hierarchies and objects have been presented in the literature [2, 25, 24]. These are primarily intended as optimizations, although they may have some value for program understanding.

Tip et al. [25] present an algorithm for slicing class hierarchies that eliminates members and inheritance relations from a C++ hierarchy. Class slicing is less powerful than specialization because it can only remove a member m from a class C if m is not used by *any* C-instance.

Sweeney and Tip [24] present an empirical study of an algorithm for detecting dead data members in C++ applications. This algorithm reports a data member to be dead if the program never reads that data member's value. This algorithm is evaluated on a set of C++ benchmark programs ranging from 600 to 58,000 lines of code. Sweeney and Tip found that up to 27.3% of the data members in the benchmarks are dead (average 12.5%), and that up to 11.6% of the object space of these applications may be occupied by dead data members at run-time (average 4.4%).

6.3 Techniques for restructuring class hierarchies

Another category of related work is that of techniques for restructuring class hierarchies for the sake of improving design, improving code reuse, and enabling reuse. Opdyke and Johnson [14, 13] present a number of behavior-preserving transformations on class hierarchies, which they refer to as *refactorings*. The goal of refactoring is to improve design and enable reuse by "factoring out" common abstractions. This involves steps such as the creation of new superclasses, moving around methods and classes in a hierarchy, and a number of similar steps. Our techniques for analyzing the usage of a class hierarchy to find design problems is in our opinion complimentary to the techniques of [14, 13].

Moore [12] presents a tool that automatically restructures inheritance hierarchies and refactors methods in Self programs. The goal of this restructuring is to maximize the sharing of expressions between methods, and the sharing of methods between objects in order to obtain smaller programs with improved code reuse. Since Moore is studying a dynamically typed language without explicit class definitions, a number of complex issues related to preserving the appropriate subtype-relationships between types of variables do not arise in his setting.

[9] Definitions 1, 3, 4, and 7 were taken from [26, 27].

7 Conclusions and Future Work

We have presented a method for finding design problems in a class hierarchy by analyzing the *usage* of the hierarchy by a set of applications. This method is based on *concept analysis* and constructs a concept lattice in which relationships between variables and class members are made explicit, and where information that members and variables have in common is "factored out". We have shown the technique to be capable of finding design anomalies such as class members that are redundant or that can be moved into a derived class. In addition, situations where it is appropriate to split a class can be detected. We have suggested how these techniques can be incorporated into interactive tools for maintaining and restructuring class hierarchies.

The present paper has focused on foundational aspects. We intend to implement an interactive class hierarchy restructuring tool based on our technique, and verify its practicality by applying it to large C++ applications. Large applications typically use libraries for which no source code is available, which will force us to make conservative assumptions in the points-to analysis. It remains to be seen to what extent this will affect the accuracy and usefulness of the resulting lattices.

We believe that there are several interesting research issues related to the question of how to present the information contained in the lattice to the user. The treatment of a number of C++ features (in particular type casts) still needs to be modeled, but we anticipate no major problems. We hope to be able to report on realistic case studies soon.

Acknowledgements

We are grateful to Robert Bowdidge, Bernhard Ganter, Christian Lindig, Peter Sweeney, and the anonymous FSE reviewers for commenting on earlier versions of this paper.

References

[1] ACCREDITED STANDARDS COMMITTEE X3, I. P. S. Working paper for draft proposed international standard for information systems—programming language C++. Doc. No. X3J16/97-0108. Draft of 25 November 1997.

[2] AGESEN, O., AND UNGAR, D. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)* (Portland, OR, 1994), pp. 355–370. *ACM SIGPLAN Notices* 29(10).

[3] BIRKHOFF, G. *Lattice Theory*. American Mathematical Society, 1940.

[4] CHOI, J.-D., BURKE, M., AND CARINI, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (1993), ACM, pp. 232–245.

[5] DAVEY, B., AND PRIESTLEY, H. *Introduction to lattices and order*. Cambridge University Press, 1990.

[6] GANTER, B., AND WILLE, R. *Formale Begriffsanalyse – Mathematische Grundlagen*. Springer Verlag, 1996.

[7] GODIN, R., AND MILI, H. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)* (Washington, DC, 1993), pp. 394–410. *ACM SIGPLAN Notices* 28(10).

[8] GODIN, R., MILI, H., MINEAU, G. W., MISSAOUI, R., ARFI, A., AND CHAU, T.-T. Design of class hierarchies based on concept (galois) lattices. *Theory and Practice of Object Systems 4*, 2 (1998), 117–134.

[9] KRONE, M., AND SNELTING, G. On the inference of configuration structures from source code. In *Proceedings of the 1994 International Conference on Software Engineering (ICSE'94)* (Sorrento, Italy, May 1994), pp. 49–57.

[10] LINDIG, C. Analyse von Softwarevarianten. Tech. Rep. 98-02, TU Braunschweig, FB Informatik, 1998.

[11] LINDIG, C., AND SNELTING, G. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)* (Boston, MA, May 1997), pp. 349–359.

[12] MOORE, I. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 235–250. *ACM SIGPLAN Notices* 31(10).

[13] OPDYKE, W., AND JOHNSON, R. Creating abstract superclasses by refactoring. In *ACM 1993 Computer Science Conference* (1993).

[14] OPDYKE, W. F. *Refactoring Object-Oriented Frameworks*. PhD thesis, University Of Illinois at Urbana-Champaign, 1992.

[15] OTT, L. M., AND THUSS, J. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering* (1989), pp. 198–204.

[16] PANDE, H. D., AND RYDER, B. G. Data-flow-based virtual function resolution. In *Proceedings of the Third International Symposium on Static Analysis (SAS'96)* (September 1996), pp. 238–254. Springer-Verlag LNCS 1145.

[17] RAMALINGAM, G., AND SRINIVASAN, H. A member lookup algorithm for C++. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation* (Las Vegas, NV, 1997), pp. 18–30.

[18] ROSSIE, J. G., AND FRIEDMAN, D. P. An algebraic semantics of subobjects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)* (Austin, TX, 1995), pp. 187–199. *ACM SIGPLAN Notices* 30(10).

[19] SHAPIRO, M., AND HORWITZ, S. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages* (Paris, France, 1997), pp. 1–14.

[20] SIFF, M., AND REPS, T. Identifying modules via concept analysis. In *Proc. International Conference on Software Maintenance* (Bari, Italy, 1997), pp. 170–179.

[21] SNELTING, G. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology 5*, 2 (April 1996), 146–189.

[22] SNELTING, G. Concept analysis – a new framework for program understanding. In *Proceeding of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Montreal, Canada, 1998), pp. 1–10.

[23] STEENSGAARD, B. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 1996), pp. 32–41.

[24] SWEENEY, P. F., AND TIP, F. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998), pp. 324–332. *ACM SIGPLAN Notices* 33(6).

[25] TIP, F., CHOI, J.-D., FIELD, J., AND RAMALINGAM, G. Slicing class hierarchies in C++. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 179–197. *ACM SIGPLAN Notices* 31(10).

[26] TIP, F., AND SWEENEY, P. Class hierarchy specialization. In *Proceedings of the Twelfth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 271–285. *ACM SIGPLAN Notices* 32(10).

[27] TIP, F., AND SWEENEY, P. F. Class hierarchy specialization. Tech. Rep. RC21111, IBM T.J. Watson Research Center, February 1998.

[28] WILLE, R. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets* (1982), 445–470.

Appendix

This appendix demonstrates that a method and its `this`-pointer will always appear together in the lattice. This fact justifies the elimination of the rows for `this`-pointers from the constructed tables.

Lemma. For any $\mathit{dcl}(C::f) \in \mathit{MemberDcls}(\mathcal{P})$, we have that:

$$\mu(\mathit{dcl}(C::f)) = \bigvee_{(x,\, \mathit{dcl}(C::f)) \in T} \gamma(x)$$

Proof. $\gamma(x) \leq \mu(\mathit{dcl}(C::f))$ for all $(x, \mathit{dcl}(C::f)) \in T$, and therefore this is also true for their supremum. On the other hand, $\mu(\mathit{dcl}(C::f))$ can not be above the supremum, because by construction of the concept lattice, $\downarrow \mu(\mathit{dcl}(C::f))$ contains exactly all x with $(x, \mathit{dcl}(C::f)) \in T$. Hence the equality holds.

Lemma. For any $\mathit{def}(C::f) \in \mathit{MemberDefs}(\mathcal{P})$, we have that:

$$\gamma(*C::f) \geq \mu(\mathit{def}(C::f))$$

Proof.

Every method call `x.f()` causes an implicit assignment `*C::f=x;` to the method's `this` pointer, hence by the assignment rule we have the implication $*C::f \rightarrow x$ and therefore $\gamma(*C::f) \geq \gamma(x)$. Furthermore, for $m = \mathit{def}(C::f)$ or $m = \mathit{dcl}(C::f)$, $\bigvee_{(x,m) \in T} \gamma(x) = \mu(\mathit{dcl}(C::f))$ by the above lemma (note that the dominance rules will enforce $\mu(\mathit{dcl}(C::f)) \geq \mu(\mathit{def}(C::f))$, hence accesses of x to f's definition do not really contribute to the supremum). Because of $\mu(\mathit{dcl}(C::f)) \geq \mu(\mathit{def}(C::f))$, we have $\gamma(*C::f) \geq \mu(\mathit{def}(C::f))$.

The last lemma shows that a method always appears below its `this`-pointer, and without the `this`-rule, they will indeed appear at different elements in the lattice if method $C::f$ does not access itself (i.e., is non-recursive).

The `this`-rule enforces $\gamma(*C::f) \leq \mu(\mathit{def}(C::f))$, and together with the lemma we may conclude the following.

Proposition. For any $\mathit{def}(C::m) \in \mathit{MemberDefs}(\mathcal{P})$ we have that:

$$\gamma(*C::f) = \mu(\mathit{def}(C::f))$$