Formal Concept Analysis and Lattices

Philipp Martis
Seminar "Verfahren zur Analyse von Programmcode"

February 27, 2015

Abstract

This paper provides an introduction to lattices and Formal Concept Analysis using lattices. Also, their applications are discussed. In particular, their application in the analysis of class hierarchies in object-oriented programming is discussed and illustrated by an example. A brief introduction to order theory is given to provide the information necessary to understand the other topics.

1 Introduction

Lattices are a very general and valuable mathematical concept laying the foundations for Formal Concept Analysis. Formal Concept Analysis in turn provides a formal model to gain insight into the structure of knowledge and information. It can help to analyze, visualize and restructure information, and find correlations and dependencies between information.

In particular, it can be used to do analysis and restructuring of class hierarchies in object-oriented programming. (Semi-)automatic tools can be built to do this job with the aid of computers [1]. Other application areas involve data in computer programs and data occurring in biology, physics, psychology, sociology and other sciences.

This article mainly deals with the applications of lattices and Formal Concept Analysis in the analysis and restructuring of class hierarchies. All topics necessary to understand this subject are covered. In particular, the basic principles of order theory are discussed and lattices and Formal Concept Analysis are explained. All topics discussed are described formally and illustrated by examples. Wherever feasible, some contextual information is provided, e.g. superficial descriptions of algorithms. Problems arising from the great expressiveness of modern programming languages and their impact on Formal Concept Analysis are also discussed briefly.

2 Order and lattice theory

To use the model of Formal Concept Analysis, one first has to understand its basic principles, especially lattices. This section provides this understanding.

First, the mathematical notion of *order* is discussed, which builds the foundation for lattices. Then, lattices are introduced, which in turn build the foundation for Formal Concept Analysis.

2.1 Order

There exist many different kinds of order in mathematics, distinguished by having or not having certain exactly defined properties. One of the most important kinds is the *partial order*:

Definition 1:

A *(partially) ordered set* $(M, \leq)$ is a set M with a relation "$\leq$" defined on it such that for all $x, y, z \in M$ [1]:

- $x \leq x$ *(reflexivity)*
- $x \leq y \ \wedge y \leq x \ \Rightarrow \ x = y$ *(antisymmetry)*
- $x \leq y \ \wedge y \leq z \ \Rightarrow \ x \leq z$ *(transitivity)*

$(M, \leq)$ is often just denoted by M. Also, instead of $(x, y) \in \leq$, usually $x \leq y$ is written. The relation "$\leq \cap \neq$" is usually denoted by "$<$" or "$\lneq$."

If, in a partially ordered set, two elements x and y are not comparable, thus $x \not\leq y \wedge y \not\leq x$, this is often denoted by

$$x \parallel y.$$

Demanding that every two elements of the set are comparable leads to the notion of a *totally ordered set*, also called a *linearly ordered set* or *chain*:

Definition 2:

A *totally ordered set* or *chain* $(M, \leq)$ is a partially ordered set with the restriction that for all $x, y \in M$

$$x \leq y \ \vee \ y \leq x$$

holds [1].

With respect to orders $\leq$ on sets P and Q (in this case, the same symbol can be used for both orders without the risk of ambiguity), maps between P and Q can be classified as follows:

Definition 3:

A map $\varphi : P \to Q$ between ordered sets $(P, \leq)$ and $(Q, \leq)$ is said to be [1]:

- *order-preserving* or *monotone* if $x \leq y$ in P implies $\varphi(x) \leq \varphi(y)$ in Q.
- an *order-embedding* if $x \leq y$ in P if and only if $\varphi(x) \leq \varphi(y)$ in Q.
- an *order-isomorphism* if it is an order-embedding which maps P onto Q.

It is easily shown that every order-embedding is injective (one-to-one), so the only difference to order-isomorphisms is that these are always surjective (map into the whole codomain).

2.1.1 Hasse diagrams

Ordered sets can be visualized in different ways. A very clean yet complete depiction is *Hasse diagrams*.

A Hasse diagram is yielded when each element x of a partially ordered set $(M, \leq)$ is represented as a point $p(x)$ in the Euclidian plane $\mathbb{R}^2$ and these points are connected according to the following rules [1]:

- Connect every two points x and $y \in M$, for which
  $$x \lneq y \ \wedge \ \neg \exists \, z \in M \colon \ x \lneq z \lneq y$$
  holds, that is points being direct "neighbours" in terms of $\leq$.

- Do this in such a way, that the line is ascending from $p(x)$ to $p(y)$ (thus, $p(x)$ has a strictly smaller second coordinate) if $x \leq y$.

Because of the transitivity of $\leq$, every two points $p(x)$, $p(y) \in \mathbb{R}^2$ with $x, y \in M \colon \ x \leq y$ either are the same point (if $x = y$) or have an ascending path between them. In both cases, $p(y)$ is said to be *above* $p(x)$ and $p(x)$ to be *below* $p(y)$.

The above rules still leave much freedom in placing the points. Doing so in an advantageous way to obtain a clear diagram can be challenging for large ordered sets and requires experience [1].

All visualizations of lattices presented in the following sections are Hasse diagrams.

2.1.2 Supremum and infimum

With the notion of order, the terms *upper bound*, *lower bound*, *supremum* and *infimum* can be defined, which are basic to the notion of a lattice:

Definition 4:

Let P be an ordered set and $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leq x$ for all $s \in S$ [1]. The term *lower bound* is defined dually.

Definition 5:

Let P be an ordered set and $S \subseteq P$. An element $x \in P$ is called the *least upper bound* or *supremum* of S if [1]

- x is an upper bound of S and
- $x \leq y$ for all upper bounds y of S.

The term *greatest lower bound* or *infimum* is defined dually.

The supremum of a set S – if any – is denoted as $\sup S$ and its infimum as $\inf S$. In the context of lattices, a more convenient notation is common [1]:

$\bigvee S$ is written instead of $\sup S$ and $\bigwedge S$ instead of $\inf S$ and they are called *join* and *meet*, respectively. The join and meet of two-element sets $\{x, y\}$ is denoted by $x \vee y$ and $x \wedge y$, respectively. To indicate that the join or meet of S is being found in a particular set P, $\bigvee_P S$ or $\bigwedge_P S$ is written.

2.2 Lattices

The core idea behind lattices is, that, if, in a partially ordered set M, not every two elements are comparable, at least the join and meet of every two elements can be determined (the definitions of these terms and their notation can be found in Section 2.1.2).

Informally, a *lattice* is an ordered set for which this restriction holds. If the join and meet of *every* subset of M can be determined, M is called a *complete lattice*.

Definition 6:

- A *lattice* is an ordered set P, such that $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$ [1]
- A *complete lattice* is an ordered set P, such that $\bigvee S$ and $\bigwedge S$ exist for all $S \subseteq P$ [1]

2.2.1 Examples

The following two examples are lattices (more precisely: Hasse diagrams of lattices) [4]:

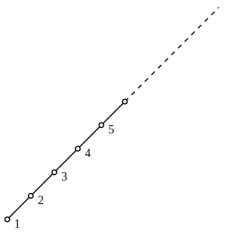

Figure 1: The natural numbers with $\leq$

The example in Figure 1 shows a Hasse diagram for $(\mathbb{N}, \leq)$. Obviously, every two elements are comparable, because there is an ascending or descending path between every two elements, as described in Section 2.1.1.

Therefore, every two elements have a join and a meet: the larger or smaller element, respectively.

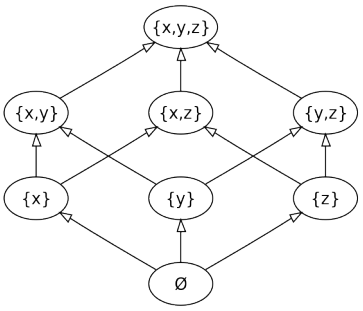

Figure 2: The powerset of $\{x, y, z\}$ with $\subseteq$

The example in Figure 2 shows a Hasse diagram of the lattice $(\mathcal{P}(M), \subseteq)$ where $M = \{x, y, z\}$ is some three-element set. Since the order is given by $\subseteq$, a subset M_1 of M is "less than or equal to" another subset M_2 if $M_1 \subseteq M_2$. The join of two subsets M_1 and M_2 of M is hence the smallest set $M_j \subseteq M$, both M_1 and M_2 are subsets of. Conversely, the meet of M_1 and M_2 is the largest set $M_m \subseteq M$, which is both a subset of M_1 and M_2.

As described in Section 2.1.1, there is an ascending path from every element m to every element which is "bigger than" (that is, a superset of) m. For example, there is an ascending path from the empty set to every other set. The sets $\{x\}$ and $\{y\}$, for example, are not comparable, so there is neither an ascending nor a descending path between them. On the other hand, every two elements have a join and a meet, so there is always an element which is reachable from these two elements by only taking ascending paths or descending paths, respectively. For $\{x, y\}$ and $\{z\}$, for example, this is $\{x, y, z\}$ and $\emptyset$, respectively.

The following example is a Hasse diagram of an ordered set which is not a lattice [4]:

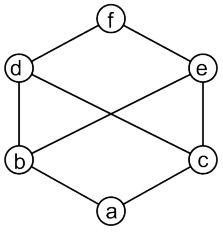

Figure 3: No lattice: $b \vee c$ does not exist, for instance

The example in Figure 3 is not a lattice. While some elements do have a join and a meet (like a and d, whose join is d and whose meet is a), some do not have both (like b and c, who have a meet, a, but no join).

2.2.2 Lattices as algebraic structures

Having the operators "$\vee$" and "$\wedge$" defined allows us to view a lattice M as an algebraic structure $\langle M, \vee, \wedge \rangle$, alternatively to viewing it as an ordered set with certain restrictions. In particular, *associativity*, *commutativity*, *idempotency* and *absorption* hold, thus, both $(M, \vee)$ and $(M, \wedge)$ are semigroups [1]:

For all $a, b, c \in M$:

- $(a \vee b) \vee c = a \vee (b \vee c)$ and $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ *(associativity)*
- $a \vee b = b \vee a$ and $a \wedge b = b \wedge a$ *(commutativity)*
- $a \vee a = a$ and $a \wedge a = a$ *(idempotency)*
- $a \vee (a \wedge b) = a$ and $a \wedge (a \vee b) = b$ *(absorption)*

A map $f : P \to Q$ is called a *lattice homomorphism* if for all $a, b \in P$

$f(a \vee b) = f(a) \vee f(b)$ and $f(a \wedge b) = f(a) \wedge f(b)$ hold.

A lattice homomorphism which is bijective is called *lattice isomorphism*. It can be shown that a map f is a lattice isomorphism iff it is an order-isomorphism [1].

3 Formal Concept Analysis

3.1 Introduction

What knowledge *is* exactly and how it can be modeled and represented is a central question in epistemology which is unlikely to be satisfiably answered. Even when restricting oneself to conceptual knowledge, one is confronted with an overwhelming number of different views and theories [2]. This makes clear that we have to focus on "some specific type of abstraction which enables us to fulfill specified aims." [2]

The following notion of a *concept* – contributed by traditional philosophy [1] – and the approach of Formal Concept Analysis provides a model to meet such aims in terms of (object-oriented) programming.

Every concept is embedded into a *context*.

Definition 7:

A *context* is a triple $(\mathcal{O}, \mathcal{A}, T)$, where $\mathcal{O}$ and $\mathcal{A}$ are sets and T is a relation between them, thus $T \subseteq \mathcal{O} \times \mathcal{A}$ [1].

As with $\leq$, usually oTa is written instead of $(o, a) \in T$.

O can be imagined as a set of objects and A as a set of attributes with oTa meaning "object o has attribute a." Also, the relation T can be imagined as a boolean table (indeed, this is one way to visualize contexts) [1].

Definition 8:

Let $(\mathcal{O}, \mathcal{A}, T)$ be a context. For $O \subseteq \mathcal{O}$ and $A \subseteq \mathcal{A}$, define [1]

- $A' := \{o \in \mathcal{O} \mid \forall a \in A : oTa\}$ and
- $O' := \{a \in \mathcal{A} \mid \forall o \in O : oTa\}$

Definition 9:

A *concept* in the context $(\mathcal{O}, \mathcal{A}, T)$ is a pair (O, A) where $O \subseteq \mathcal{O}$, $A \subseteq \mathcal{A}$, $O' = A$ and $A' = O$ [1].

Given a concept (O, A), O is called the *extent* and A is called the *intent* of the concept [1].

Within this notion, a concept is an object-attribute pair with A being all the attributes all objects in O have, while O is all the objects having all the attributes in A.

The set of all concepts of a context $(\mathcal{O}, \mathcal{A}, T)$ is denoted $\mathfrak{B}(\mathcal{O}, \mathcal{A}, T)$ ($\mathfrak{B}$ comes from the German word "Begriff") [1].

Lemma:

Given a context $(\mathcal{O}, \mathcal{A}, T)$, an index set J and sets $O, O_j \subseteq \mathcal{O}$ for $j \in J$ and $A, A_j \subseteq \mathcal{A}$ for $j \in J$, the following statements hold [1]:

- $O \subseteq O''$ and $A \subseteq A''$
- $O_1 \subseteq O_2 \Rightarrow O_1' \supseteq O_2'$ and $A_1 \subseteq A_2 \Rightarrow A_1' \supseteq A_2'$
- $O' = O'''$ and $A' = A'''$
- $(\bigcup_{j \in J} O_j)' = \bigcap_{j \in J} O_j'$ and $(\bigcup_{j \in J} A_j)' = \bigcap_{j \in J} A_j'$
- $O \subseteq A' \Leftrightarrow O' \supseteq A$

3.2 Link to lattices

3.2.1 The ordering of concepts

Given a context $(\mathcal{O}, \mathcal{A}, T)$, an order $\leq$ can be defined by
$(O_1, A_1) \leq (O_2, A_2) :\Leftrightarrow O_1 \subseteq O_2 \Leftrightarrow A_1 \supseteq A_2$ [1]

It is easily seen that $\leq$ defines a partial order on $\mathfrak{B}(\mathcal{O}, \mathcal{A}, T)$ – reflexivity, transitivity and antisymmetry are inherited from $\subseteq$. Given that order, contexts can be visualized using Hasse diagrams.

3.2.2 Concept Lattices

$(\mathfrak{B}(\mathcal{O}, \mathcal{A}, T), \leq)$ is a complete lattice, called the *Concept Lattice* of the context $(\mathcal{O}, \mathcal{A}, T)$ [1].

3.3 Visualization of contexts

There are at least three natural ways to represent a context $(\mathcal{O}, \mathcal{A}, T)$ visually [1]:

- As a so-called *cross table* with fields indicating whether an object o has an attribute or not. Most often, the objects are put in the rows of the table and the attributes in its columns.
- As a bipartite graph $G = (V, E)$ with $V = \mathcal{O} \cup \mathcal{A}$ and $E = T$.
- As a Hasse diagram of $(\mathfrak{B}(\mathcal{O}, \mathcal{A}, T), \leq)$, which, as explained in Section 3.2.2, is a lattice.

The latter solution has the advantage that it can be used even for big contexts and that it illustrates important information explicitly [1]. In particular, it reveals hierarchies implicated by $\leq$ as defined in Section 3.2.1.

This can be of great use and is part of the power of Formal Concept Analysis, as will be demonstrated in Section 4 by means of an example class hierarchy.

3.4 Implications

Implications between attributes are an important notion within Formal Concept Analysis. They are valuable in the construction process of the object-attribute relation T or if additional information about relations between objects and attributes shall be provided [3].

Let $A_1, A_2 \in \mathcal{A}$ be sets of attributes. A_1 is said to imply A_2 if all objects having all the attributes in A_1 also have all attributes in A_2 [3]. This is denoted by $A_1 \rightarrow A_2$.

Definition 10:

In a context $(\mathcal{O}, \mathcal{A}, T)$, $A_1 \subseteq \mathcal{A}$ *implies* $A_2 \subseteq \mathcal{A}$ if for all $o \in \mathcal{O}$
$\forall a_1 \in A_1 : oTa_1 \Rightarrow \forall a_2 \in A_2 : oTa_2$ [3]

4 Analyzing class hierarchies using Formal Concept Analysis

The design of modern information systems – possibly containing millions of lines of source code – is an art and requires special care if the system is designed decentrally by many people – as in open source systems – or is a library whose operational area its designer does not even know of [5].

Formal Concept Analysis provides help to analyze such complex systems and find out about certain inconsistencies, such as:

- Unused class members, possibly indicating that they should be removed or moved to a derived class [3]
- Different subsets of class members used separately in a program, possibly indicating that the class should be split up into multiple classes [3]

The aim of Formal Concept Analysis is also to be the basis of (semi-)automatic tools aiding the programmer in carrying out such restructuring [3].

Formal Concept Analysis can be done in several ways: analyzing the class hierarchy in isolation or together with a set of programs using it [3]. In the latter case, how the programs use the class hierarchy can be viewed for each program in isolation or as an aggregate view [3].

4.1 The example class hierarchy

Consider the following C++ example [3]:

```
class A {
 public:
  virtual int f() { return g(); }
  virtual int g() { return x; }
  int x;
};

class B : public A {
 public:
  int g() override { return y; }
  int y;
};

class C : public B {
 public:
  int f() override { 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();
}
```

Figure 4: Example class hierarchy

4.2 Applying Formal Concept Analysis

This section gives a detailed view about how Formal Concept Analysis is used for gaining insight into the structure of a class hierarchy and possibly getting recommendations for a restructured class hierarchy. The example class hierarchy shown in Figure 4 is used to illustrate the procedure.

4.2.1 Determining the context

To apply Formal Concept Analysis to the class hierarchy to be analyzed, the context $(\mathcal{O}, \mathcal{A}, T)$ first has to be defined properly.

Basically, the objects shall represent the variables of user defined types or of type "pointer to user defined type" and the attributes the members of user defined types. T would then reflect which variable (or more precisely, which object in terms of programming) uses which of its members.

Due to the fact that variables can be assigned to each other, so a variable does not always refer to the same object, and programming language constructs such as virtual functions, the algorithm for determining the context becomes rather complicated. Nevertheless, it shall be described here in some detail.

First, some auxiliary definitions are made, which are all fairly self-explanatory:

Definition 11:

Given a program or set of programs $\mathcal{P}$, the following can be defined [3]:

- $ClassVars(\mathcal{P}) := \{v \mid v$ is a variable of a user defined type$\}$
- $ClassPtrVars(\mathcal{P}) := \{p \mid p$ is a variable of type "pointer to user defined type"$\}$
- $MemberDcls(\mathcal{P}) := \{dcl(C::m) \mid m$ is a data member or virtual method in class $C\}$
- $MemberDefs(\mathcal{P}) := \{def(C::m) \mid m$ is a virtual or nonvirtual method in class $C\}$
- $MayPointTo(\mathcal{P}) := \{(p, v) \mid v \in ClassVars(\mathcal{P}), p \in ClassPtrVars(\mathcal{P}),$ p may point to $v\}$
- $MemberAccess(\mathcal{P}) := \{(m, v) \mid$ `v.m` occurs in $\mathcal{P}$, m is a class member, $v \in ClassVars(\mathcal{P})\}$ $\cup \{(m, *p) \mid$ `p->m` occurs in $\mathcal{P}$, m is a class member, $p \in ClassPtrVars(\mathcal{P})\}$ $\cup \{(f, y) \mid$ `p->f` occurs in $\mathcal{P}$, $(p, y) \in MayPointTo(\mathcal{P})$, f is a virtual method$\}$

- $Assignments(\mathcal{P}) \coloneqq \{(v, w) \mid$ `v = w` occurs in $\mathcal{P}$, $v, w \in ClassVars(\mathcal{P})\}$
  $\cup \{(*p, w) \mid$ `p = &w` occurs in $\mathcal{P}$, $p \in ClassPtrVars(\mathcal{P}), w \in ClassVars(\mathcal{P})\}$
  $\cup \{(*p, *q) \mid$ `p = q` occurs in $\mathcal{P}$, $p, q \in ClassPtrVars(\mathcal{P})\}$
  $\cup \{(*p, w) \mid$ `*p = w` occurs in $\mathcal{P}$, $p \in ClassPtrVars(\mathcal{P})$, $w \in ClassVars(\mathcal{P})\}$
  $\cup \{(v, *q) \mid$ `v = *q` occurs in $\mathcal{P}$, $v \in ClassVars(\mathcal{P})$, $q \in ClassPtrVars(\mathcal{P})$
  $\cup \{(*p, *q) \mid$ `*p = *q` occurs in $\mathcal{P}$, $p, q \in ClassPtrVars(\mathcal{P})\}$

To model the virtual function call mechanism accurately, declarations and definitions of virtual functions are distinguished, since a calling function does not have to know the body definition of a virtual function it calls [3].

$ClassPtrVars$ contains implicitly declared `this`-pointers of functions [3], because track has to be kept of which object a function call was invoked on. To distinguish the `this`-pointers of different functions, they are referred to using the name of the respective function, e.g. the `this`-pointer of `A::f()` is referred to as `A::f`.

Data members are modeled as declarations because they do not have a `this`-pointer allowing access of other members associated with them [3].

Naming our example program from Figure 4 "$\mathcal{P}_1$", the following sets are obtained [3]:

- $ClassVars(\mathcal{P}_1) = \{a, b, c\}$
- $ClassPtrVars(\mathcal{P}_1) = \{ap, A::f, A::g, B::g, C::f\}$
- $MemberDcls(\mathcal{P}_1) = \{dcl(A::f), dcl(A::g), dcl(A::x), dcl(B::g), dcl(B::y), dcl(C::f), dcl(C::z)\}$
- $MemberDefs(\mathcal{P}_1) = \{def(A::f), def(A::g), def(B::g), def(C::f)\}$
- $MayPointTo(\mathcal{P}_1) = \{(ap, a), (ap, b), (ap, c), (A::f, a), (A::f, b), (C::f, c), (A::g, a), (B::g, b), (B::g, c)\}$
- $MemberAccess(\mathcal{P}_1) = \{(x, *A::g), (y, *B::g), (z, *C::f), (g, *A::f), (g, *C::f), (f, *ap), (f, a), (f, b), (f, c), (g, a), (g, b), (g, c)\}$

Let $\mathcal{P}$ be a given program. Then $\mathcal{O}$, $\mathcal{A}$ and T are built up successively as follows [3]:

The objects and attributes are the variables and members, respectively, thus
$\mathcal{O} = ClassVars(\mathcal{P}) \cup ClassPtrVars(\mathcal{P})$ and
$\mathcal{A} = MemberDcls(\mathcal{P}) \cup MemberDefs(\mathcal{P})$

As described in Section 3.3, usually the elements of $\mathcal{O}$ become the rows and the elements of $\mathcal{A}$ become the columns of the table T. Building T is accomplished by an iterative approach, which is repeated until no more changes take place in T.

The first step in building the table is to add all obvious member accesses, as written in the code, to it. For data members, the member and the variable accessing it are added to T; similarly for nonvirtual member functions. For virtual functions, their declaration, that is, $dcl(...)$, is added to T if it is called through a pointer or reference, and their definition, $def(...)$, otherwise. Note that virtual functions, if not called through a pointer or reference, act just like normal member functions.

Next, for each function $C::f()$, the object it is invoked on, given in terms of a `this`-pointer $*C::f$, and its definition $def(C::f)$ are added as a pair to the table, thereby guaranteeing that the definition will appear above or as the same node as the object in the resulting lattice. If this is not done, the natural notion that an object uses a member function it invokes would only materialize in the table for functions which call themselves (recursive functions). As will later become clear, the members used by an object will always appear above it in the lattice. It is guaranteed anyway that $def(C :: f)$ will appear below or at the same level as the object $*C :: f$ in the resulting lattice [3]. Both guarantees together ensure that both notions will appear as the same lattice element, allowing us to later remove the objects representing `this`-pointers from the table again.

When variables are assigned, the object assigned becomes the object associated with the variable being assigned to and thus can access everything the object pre-

viously associated with the variable could access. Thus, assignment is an implication in terms of Formal Concept Analysis. Therefore, the next rule when constructing T is to add these implications between objects: every attribute the object in the "source row" has is also associated with the object in the "target row".

As the last iteration step, *dominance* or *hiding* effects have to be taken into account. If two data or nonvirtual function members have the same name and are declared in classes of the same hierarchy, the member in the derived class hides the member in the base class. Since potential suggestions for a new class hierarchy must preserve the semantics of the member accesses occurring in the code, these hiding relationships must be preserved. Formally, such a relationship is nothing more than an implication between attributes (columns), analogously to the implication between objects. Hence, to reflect that implication, the columns representing dominating attributes are copied to the columns representing dominated attributes.

As mentioned previously, these steps are repeated until T doesn't change anymore and then the rows representing `this`-pointers are erased from T.

Background knowledge that is not reflected in the table, for example desired hiding effects currently not expressed in the source code, can be provided as additional implications [3].

4.2.2 Modeling constructors

Special attention has to be payed to constructors. In C++, for instance, the compiler will generate default constructors, that is, constructors taking no arguments, under certain conditions, which in turn call the default constructors of all base classes and certain members [5]. Furthermore, the compiler will generate calls to constructors in certain cases [3], for example when an object is created on the stack, on the heap or as a member of another object.

There are different approaches to modeling constructors and constructor calls, each persuing specific aims. The approach used in the original form of the modeling is to leave compiler generated constructors and constructor calls – and only those – out of account [3]. This eliminates the illusion that members constructed by a compiler-generated call to a default constructor are actually accessed [3].

	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	×	×				×			×	×

Figure 5: The final cross table for $\mathcal{P}_1$

Alternatively, all constructors and constructor calls can be left out of account, which, for example, is useful for finding unused members.

Taking all constructors and constructor calls into account can also be useful. Doing so makes it possible to check if all members are initialized.

4.2.3 Building the lattice

To construct the lattice out of the final table T, Ganter's algorithm can be used, which is exponential in the worst case, but has a typical complexity of $\mathcal{O}(n^3)$ [3]. It can be shown that exponential behavior is extremely rare, though [3].

The final cross table for the example program $\mathcal{P}_1$ from Section 4.1 is shown in Figure 5, and its final lattice in Figure 6.

4.2.4 Reading a lattice

Lattices, especially when visualized as Hasse diagrams, give a clean depiction of the structure they represent. Although they contain all information about the relations between objects and attributes, they are easy to read. As described in Section 2.1.1, the terms "above" and "below"

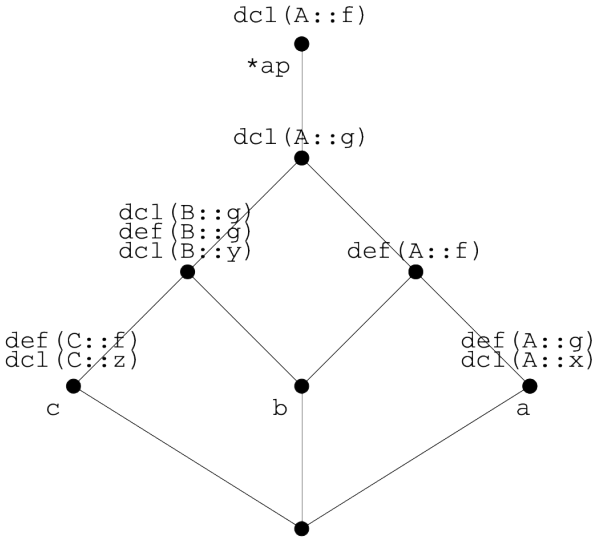

Figure 6: The final lattice of $\mathcal{P}_1$

are used to describe that two nodes are the same or have an ascending or descending, path between them, respectively. The most important relationships to be read from a lattice and how this is done are described in the following:

- Members appear above the objects that access them (or may access them, at least) [3].
- Members not accessed at all would appear at the bottom element of the lattice [3].
- Objects not accessing any of their members would appear at the top element of the lattice [3].
- Data members of a base class `B` that are not used by (instances of) all derived classes of `B` appear above some but not all derived classes of `B` [3].
- If constructors are modeled (see Section 4.2.2), they appear below the data members initialized by them [3].
- Variables of the same type accessing different subsets of their members appear as different points in the lattice [3].
- If a function is overridden in a derived class, its declaration and its definition appear as different points in the lattice. Each definition then will appear above the variables representing the objects that use it.

Colors and names for the lattice elements can be used to make the lattice more readable [3].

With regard to the example program $\mathcal{P}_1$ from Section 4.1, the following facts are revealed in the lattice (shown in Figure 6):

- No members are not accessed at all – the bottom node of the lattice does not contain any member names.
- All objects access some of their members – the top node of the lattice does not contain any variable names.
- The function declaration `A::f()` may be used by every object – its name appears at the top node of the lattice.
- The data member `A::x` is used by the object `a` only and not by the objects of the derived classes – its name appears above `a` only.
- The function definition `A::f()` may be used by `a` and `b`, but not by `c` (since class `C` has its own definition of `f()` overriding `A::f()`) – its name appears above `a` and `b`, but not above `c`.

Since $\mathcal{P}_1$ is an artificial and short example program, its usefulness for illustrating which possibilities for refining the class hierarchy can be read from a lattice is rather limited. Nevertheless, some aspects are revealed in the diagram:

- The classes in the hierarchy are relatively tightly coupled; they cannot be simply split up into multiple class hierarchies. This is primarily caused by `b`, which uses members of both class `A` and class `B`, while class `B`'s members are also used by `c`.
- Since class `A`'s data member is not used by objects of derived classes, class `A` does not play the typical role of a (non-abstract) base class. If its member function `A::f()` does not really elicit polymorphic behavior of some common aspect by calling `g()`, the function `f()` may be split up into two distinct functions

`A::f()` and `B::f()`, thereby separating class `A` from the class hierarchy and potentially greatly simplifying it.

4.3 Possibilities for expansion

As discussed here, Formal Concept Analysis used for analyzing class hierarchies is by itself a general and valuable concept which reveals many important facts about the hierarchies.

As discussed in Section 4.2.2, adaptions can be made to fit different use cases. In addition, it can be refined and expanded to take issues like runtime and design concerns into account. For example:

- Several cross tables representing different kinds of access operations (cached vs. uncached, checked vs. unchecked etc.) can be used. Performance and stability concerns can be addressed by analysing these tables separately. For example it can be verified that uncached accesses occur only in debug code.
- Access restrictions refining the usual `public` / `private` / `protected` scheme can be formulated and checked. For example, it may be stated that a member may only be modified in certain source files.
- Small extensions like being able to mark attributes as "deprecated" can be used to make Formal Concept Analysis helpful for the transition of code.

5 Conclusion

Lattices and Formal Concept Analysis are very general and valuable concepts. They are very useful for revealing information about the structure of class hierarchies in object-oriented programming. More generally, they are useful in every application where dependencies and other relations between entities play an important role. Using Hasse diagrams, these relations can be presented in a clear and concise way.

Additionally, Formal Concept Analysis provides the foundation for algorithms doing semi-automatic or automatic restructuring of the class hierarchy or information.

The approach can be used in a flexible way to adapt it to several use cases. In terms of analyzing class hierarchies, many opportunities exist to refine and expand it. This allows to make it more flexible and powerful and to make it cover interesting topics like runtime concerns.

References

[1] B.A. Davey and H.A. Priestley, in *Introduction to Lattices and Order*, 2nd ed. Cambridge, England: Cambridge University Press, 2002, pp. 2-76.

[2] R. Wille, "Concept Lattices and Conceptual Knowledge Systems", *Computers Math. Applic.*, vol. 23, no. 6-9, pp. 493-515, 1992. Available: `http://dx.doi.org/10.1016/0898-1221(92)90120-7`

[3] G. Snelting, "Understanding Class Hierarchies Using Concept Analysis", *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 3, pp. 540-582, 2000

[4] "Verband (Mathematik)" in Wikipedia - Die freie Enzyklopädie. Available: `http://de.wikipedia.org/wiki/Verband_(Mathematik)`

[5] B. Stroustrup, *The C++ Programming Language*, 4th ed. Boston, USA: Addison Wesley, 2014