Fast Concept Analysis

Christian Lindig
Harvard University
Division of Engineering and Applied Sciences
Cambridge, Massachusetts
lindig@eecs.harvard.edu

April 21, 2002

Abstract

Formal concept analysis is increasingly used for large contexts that are built by programs. This paper presents an efficient algorithm for concept analysis that computes concepts together with their explicit lattice structure. An experimental evaluation uses randomly generated contexts to compare the running time of the presented algorithm with two other algorithms. Running time increases quadratically with the number of concepts, but with a small quadratic component. At least contexts with sparsely filled context tables cause concept lattices grow quadratically with respect to the size of their base relation. The growth rate is controlled by the density of context tables. Modest growth combined with efficient algorithms lead to fast concept analysis.

1 Introduction

Concept analysis has proven to be a valuable tool for gaining insight into complex data [5, 16]. In many applications of concept analysis experts learn from formal contexts by inspecting their carefully layed out concept lattices. Contexts in these applications tend to have a modest size. Otherwise, the resulting concept lattices are hard to analyze visually, even with support of tools like Toscana [17]. The algorithmic complexity of concept analysis for these applications is consequently a minor concern. But concept analysis is used increasingly for applications like program analysis inside a compiler or is combined with statistical analysis where large contexts are constructed by a program [14, 15, 11]. The resulting concept lattice is no longer inspected visually but is part of an application's internal data structure. For these

applications the algorithmic complexity of concept analysis does matter. This paper presents an efficient algorithm for computing the concept lattice and some empirical complexity results.

2 Computing the Concept Lattice

Ganter's algorithm NextConcept is probably the best known algorithm for concept analysis [3, 4]. It computes all concepts $L(\mathcal{G}, \mathcal{M}, \mathcal{I})$ from a context $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ in a total lectical order. The algorithm's asymptotic running time $O(|\mathcal{G}|^2 \times |\mathcal{M}| \times |L(\mathcal{G}, \mathcal{M}, \mathcal{I})|)$ depends only linearly on the size of the concept lattice. Most applications of concept analysis not only use the concepts (G, M) from $L(\mathcal{G}, \mathcal{M}, \mathcal{I})$ but their lattice structure as well. The explicit lattice structure is made up from the upper and lower neighbors of each concept. The lattice structure itself is not immediately available from Ganter's algorithm (and the set of all concepts) since it is an implicit property of concepts. So most applications of concept analysis need an algorithm that computes both concepts and their explicit lattice structure.

For a finite context $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ all concepts and their lattice structure can be computed by finding all upper neighbors of a concept: starting from a known concept, a small set of greater concepts can be computed that is known to include all upper neighbors. These can be identified by an efficient test. Starting at the well known smallest concept of a lattice the algorithm recursively computes all concepts and their lattice structure.

Given a concept (G, M) that is distinct from the maximum $\top$ of L, the following set S contains concepts greater than (G, M):

$$S = \{((G \cup \{g\})'', (G \cup \{g\})') \mid g \notin G\}$$

Every g not already part of G *generates* a concept $((G \cup \{g\})'', (G \cup \{g\})')$. The set S of all such concepts contains at most $|\mathcal{G}|$ members; each member is greater than (G, M) but not necessarily an upper neighbor. Theorem 1 identifies the upper neighbors of (G, M) among the concepts in S.

Theorem 1 *Let be $(G, M) \in L(\mathcal{G}\,\mathcal{M}, \mathcal{I})$ and $(G, M) \neq \top$. Then $(G \cup \{g\})''$, where $g \in \mathcal{G} \setminus G$, is an extent of an upper neighbors of (G, M) if and only if for all $y \in (G \cup \{g\})'' \setminus G$ the following holds: $(G \cup \{y\})'' = (G \cup \{g\})''$. A proof can be found in [11].*

Monotony of $''$ ensures that every extent in S is a super-set of the enlarged extent: $G \cup \{g\} \subseteq (G \cup \{g\})''$. For an upper neighbor all elements

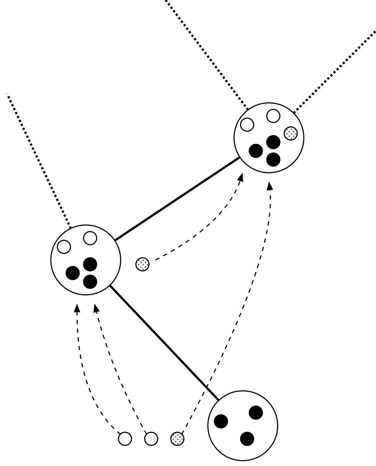

Figure 1: Sketch for Theorem 1. Large circles represent concepts that include their extents as small circles. An arrow between an object outside of a concepts and a greater concept visualizes the generation of the greater concept by adding the object to the concept next to it. Solid lines between concepts denote their lattice structure.

added by the $''$-Operator $(G \cup \{g\})'' \setminus (G \cup \{g\})$ themselves generate this extent. The principle is illustrated by the extents of three concepts shown in Fig. 1: adding any of the two white objects at the bottom to the extent of the concept to their right yields the extent on the left. It includes all (black) objects from the bottom concept plus the two white objects. This extent belongs to a upper neighbor of the bottom concept because every new (white) object generates the concept. Adding the grey object to the concept at the bottom yields the top extent. It includes the grey object and also the white objects inherited from the left concept. Thus it does not belong to a an upper neighbor because some of its objects generate a different concept.

When the grey object is added to the concept on the left it again generates the concept at the top. This time the top concept is is an upper neighbor because the grey object is the only new one and it generates the concept that it is part of.

The observation from Theorem 1 is captured by Algorithm NEIGHBORS in Fig. 2 that computes the upper neighbors of (G, M). For every concept generated by $g \in \mathcal{G} \setminus G$ it tests all elements added by the $''$-operator. Whenever an element is found that (a) is different from g, and (b) is not part of the initial G, and (c) itself generates an upper neighbor (as recorded in

```
NEIGHBORS ((G, M), (𝒢, ℳ, ℐ))
 1  min ← 𝒢 \ G
 2  neighbors ← ∅
 3  foreach g ∈ 𝒢 \ G do
 4     M_1 ← (G ∪ {g})'
 5     G_1 ← M_1'
 6     if ((min ∩ (G_1 \ G \ {g})) = ∅) then
 7        neighbors ← neigbors ∪ {(G_1, M_1)}
 8     else
 9        min ← min \ {g}
10  return neighbors
```

Figure 2: Algorithm NEIGHBORS computes the upper neighbors of a concept.

`min`), the actual extent may not belong to an upper neighbor; in that case g is removed from `min`.

The meaning of the set `min` used by the algorithm is a bit difficult to grasp: it contains elements from $\mathcal{G} \setminus G$ that generate upper neighbors. Initially all elements are assumed to generate neighbors and later elements are removed that possibly do not. At the end of the algorithm `min` is a minimal set of elements that generate the complete set of upper neighbors.

The algorithm is best understood by the means of an example: assume that G_1 is the extent of an upper neighbor of (G, M). Both x and y generate G_1 and are considered by the algorithm in that order. Initially both x and y are members of `min`. First all members of G_1 different from x—y is among them—are checked against `min`: y is found in `min` and so x is (falsely) assumed *not* to generate an upper neighbor and is removed from `min`. Next all elements (x) different from y are checked: x is no longer in `min` and thus the concept generated by y is known to be an upper neighbor. Whenever a neighbor is generated by a number of elements from $\mathcal{G} \setminus G$, only the last one considered by the algorithm is detected as neighbor-generating and therefore stays in `min`.

The asymptotic complexity of Algorithm NEIGHBORS is $O(|\mathcal{G}|^2 \times |\mathcal{M}|)$: computing the hull using the $''$-operator takes $O(|\mathcal{G}| \times |\mathcal{M}|)$ and is required $|\mathcal{G}|$ times when G is empty.

Algorithm NEIGHBORS can be employed to recursively compute all concepts L of a context by starting from the smallest concept $(\emptyset'', \emptyset')$ of the lattice; the resulting Algorithm LATTICE is shown in Fig. 3. Every concept c has two lists associated with it: the list c^* of its upper neighbors and the

```
LATTICE (𝒢, ℳ, ℐ)
 1  c ← (∅″, ∅′)
 2  insert (c, L)
 3  loop
 4     foreach x in NEIGHBORS (c, (𝒢, ℳ, ℐ))
 5        try x ← lookup (x, L)
 6        with NotFound → insert (x, L)
 7        x_* ← x_* ∪ {c}
 8        c^* ← c^* ∪ {x}
 9     try c ← next (c, L)
10     with NotFound → exit
11  return L
```

Figure 3: Algorithm LATTICE computes the concept lattice of $(\mathcal{G}, \mathcal{M}, \mathcal{I})$.

list c_* of its lower neighbors.

One concept may be shared by two different concepts as their upper neighbor. While the algorithm processes each of the two concepts their shared upper neighbor must be detected in order to get the relationships right. For this purpose all concepts are stored in a search tree L [2]. Each time the algorithm finds a neighbor it searches it (using `lookup`) in the tree L to find previously inserted instances of that concept. In case the concept is found, the existing lists of neighbors are updated; otherwise the previously unknown concept is entered into the tree.

The algorithm inserts concepts into L and looks them up at the same time: `next`(c, L) asks for the smallest concept that is greater than c with respect to the total order $\prec$ used inside the tree. To make sure all concepts that are inserted are also considered for their upper neighbors, the total tree order $\prec$ must relate to the partial lattice order $\leq$ in the following way: $c_1 < c_2$ implies $c_1 \prec c_2$. This way recently inserted neighbors are greater than the actual concept with respect to $\prec$ and will be considered later by `next`. The lectical order defined by Ganter for Algorithm NEXTCONCEPT can be used as tree order $\prec$ for this purpose.

The asymptotic worst case complexity of Algorithm LATTICE is $O(|L| \times |\mathcal{G}|^2 \times |\mathcal{M}|)$ since the operations on the search tree do not add to the complexity. It thus has the same asymptotic complexity as Ganter's NEXTCONCEPT algorithm.

Table 1: Some figures about the randomly generated contexts.

Parameter	min	max	avrg	std dev
context size $\lvert\mathcal{I}\rvert$	2	1 572	209.7	229.9
lattice size $\lvert L\rvert$	1	11 148	330.7	896.0
context fill ratio	0.01	1.00	0.17	0.15

3 Evaluation

For evaluation NextConcept and Lattice were implemented and applied to randomly generated contexts. The implementation used the Objective Caml system, a functional programming language and dialect of Standard ML that is known for emitting effective native code [8, 12]. All tests were ran on an otherwise idle 200 MHz AMD K6 Linux 2.0 system.

The experiment used 1 000 randomly generated contexts $(\mathcal{G}, \mathcal{M}, \mathcal{I})$. Their corresponding context tables were between 1×2 and 81×81 elements in size, where $\mathcal{I}$ contained up to 1 572 elements. These figures together with their averages and standard deviations are shown in table 1. A context table holds up to $|\mathcal{G}| \times |\mathcal{M}|$ elements; the size of the relation $\mathcal{I}$ with respect to the maximum size is called the *context fill ratio* and is the quotient of $|\mathcal{I}|$ and $|\mathcal{G}| \times |\mathcal{M}|$. The random process created small contexts with small fill ratios more frequently than other contexts; both parameters contribute to small concept lattices as we will see below.

For each randomly generated context the corresponding concept lattice was computed using NextConcept and Lattice. Additionally, a third Algorithm Concepts was applied; it is used in the author's program `concepts` which has gained some popularity for computing concept lattices in the past [10]. It was re-implemented in Objective Caml and utilizes Algorithm NextConcept internally. Algorithm Concepts detects the lattice structure by post-processing the output of Algorithm NextConcept which takes $O(|L|^2)$ time. It basically implements the Algorithm Ganter-Alaoui as described by Godin et al. in [6]. For all algorithms the CPU time spent versus the lattice size is shown in Fig. 4.

Although the overall lattice structure has some impact on the running time of all algorithms, running time depends foremost on lattice size $|L|$: Fig. 4 shows polynomial approximations for this dependency, which were gained from the least-square method; coefficients for all approximations can be found in table 2.

Algorithm NextConcept turned out to be the most effective which comes as no surprise, since it computes the set of all concepts only. Com-

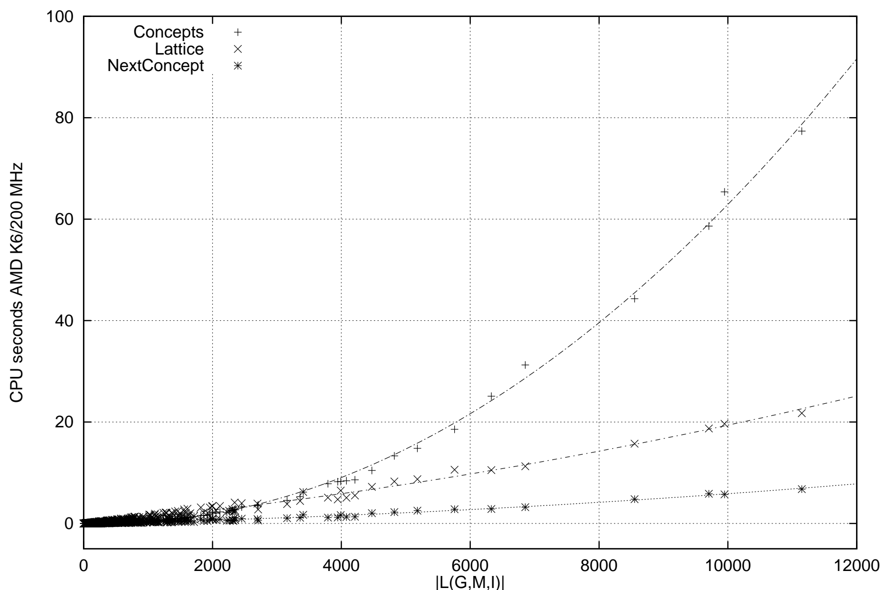

Figure 4: Running time of algorithms versus lattice size.

Table 2: Coefficients of polynomials $t = a_2|L|^2 + a_1|L| + a_0$ for approximating runtime measured in AMD K6 200 CPU seconds.

Algorithm	a_2	a_1	a_0
Concepts	6.721×10^{-7}	-0.00043	0.04699
Lattice	7.764×10^{-8}	0.00116	-0.03158
NextConcept	3.255×10^{-8}	0.00026	0.00101

puting concepts as well as their lattice structure takes twice as long for the same input when Algorithm Lattice is used. The runtime of both algorithms increases quadratically with the size of the concept lattices but the quadratic component is small. Thus, they both result in fast concept analysis. This stands in contrast to Algorithm Concepts's runtime that increases also quadratically but at a much larger rate.

Since most applications of concept analysis need the lattice structure of concepts explicitly Algorithm Lattice is a natural choice. As it is easy to implement it could be considered even when the lattice structure is not required. A minor drawback compared with Algorithm NextConcept is memory utilization: while Algorithm NextConcept computes a concept solely from a single predecessor, Algorithm Lattice stores all concepts in

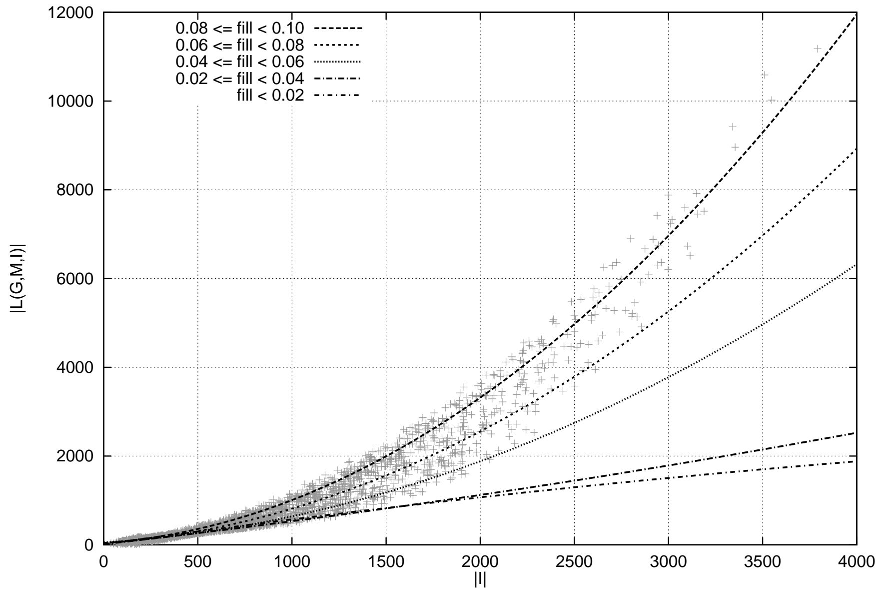

Figure 5: Lattice size $|L|$ versus context size $|\mathcal{I}|$.

a (main memory) persistent search tree.

4 Size of Concept Lattices

Concept lattices can grow exponentially in size with respect to their contexts [5]. However, this happens rarely in practical applications where concept lattices tend to be much smaller. Since running times of the algorithms presented here depend on lattice size, the actual size of concept lattices is of some interest. It has been investigated with an experiment that aimed to identify those context parameters controlling lattice size.

The experiment contained 3 187 pairs of a randomly generated context $(\mathcal{G}, \mathcal{M}, \mathcal{I})$, and its concept lattice. Context table sizes $|\mathcal{G}| \times |\mathcal{M}|$ varied between 19×87 and 201×193, relation sizes $|\mathcal{I}|$ between 38 and 3 792. Contexts were roughly equally distributed with respect to $|\mathcal{G}|$, table side length ratio $\min(|\mathcal{G}|/|\mathcal{M}|, |\mathcal{M}|/|\mathcal{G}|)$, and context fill ratio. Small relations (where size is measured by $|\mathcal{I}|$) were more frequent than large ones. All contexts had a fill ratio below 0.1 and therefore had sparsely filled context tables. The experiment was restricted to these contexts because it was originally carried out for component libraries where sparse contexts are common [9, 11].

Table 3: Approximations of lattice size $|L| = a_2|\mathcal{I}|^2 + a_1|\mathcal{I}| + a_0$ for different context classes. Parameter n denotes the class cardinality.

fill	a_2	a_1	a_0	n	$\min|\mathcal{I}|$	$\max|\mathcal{I}|$
0.00…0.02	-3.0688×10^{-5}	0.58955	13.776	357	70	592
0.02…0.04	3.6527×10^{-5}	0.48174	14.153	599	38	1282
0.04…0.06	0.00032621	0.26237	52.876	685	51	2286
0.08…0.08	0.00048146	0.29659	36.275	697	74	2856
0.08…0.10	0.00067034	0.29714	40.730	849	105	3792

While looking for functional relations between a context and its number of concepts, different context parameters were considered. The context parameter found to predict the size of a lattice best was context size $|\mathcal{I}|$: Fig. 5 shows the observed lattice size versus context size; each dot in the graph represents a single experiment. At least for sparse contexts lattice, size does not grow exponentially but quadratically with respect to context size $|\mathcal{I}|$.

As a second parameter, the context fill ratio $|\mathcal{I}|/(|\mathcal{G}| \times |\mathcal{M}|)$ was found to have some impact on the lattice size. Coloring the dots in Fig. 5 according to their context fill ratio would yield distinctly colored areas. The figure instead shows polynomial approximations for five different classes of contexts, where classes are based on fill ratios. Table 3 shows the classes and the approximations found for contexts in this classes. The largest concept lattices were observed for large contexts that had also a high fill ratio, which means their context table were relatively dense.

5 Comparison

Algorithms to compute concept lattices and the size of concept lattices have been studied before

[13, 6, 1, 4]. Although the two algorithms presented here do not lead to improved complexity results, they are based on a new theorem that leads to a remarkable simple implementation of concept analysis.

Godin et al. presented an incremental algorithm to recompute the concept lattice after adding a new object $g \in \mathcal{G}$ and a set $M \subseteq \mathcal{M}$ of related attributes [6]. Due to the incremental aspects, their results are not directly comparable. They found the running time of their algorithm to be quadratically increasing at a slow rate with respect to the size of $|\mathcal{G}|$. While comparing their algorithm with one similar to Algorithm Concepts they also noted that post-processing the output of NextConcept (to get the

lattice structure) is inefficient.

Godin et al. and Carpineto at al. investigated the size of concept lattices. They used both real live data for experiments as well as uniformly distributed data similar to the experiments in section 4. Although real live data give important insight into an algorithm's behavior they are difficult to reproduce. Because the size of lattices depends on may variables even benchmarks are hard to design. When only a small number of variables is chosen to predict lattice size, the classes of equivalent contexts are large. This is honored by our experiments which use a much larger number of test cases than earlier experiments.

Both Godin et al. and Carpineto et al. considered lattice size as a function of the total number of objects $|\mathcal{G}|$, the total number of attributes $|\mathcal{M}|$, and the average or fixed number of attributes per object. We found the context size $|\mathcal{I}|$ to be the primary variable the lattice size depends on. We therefore propose to use it for future experiments. However, the *context fill ratio* that we found as a secondary variable relates to the notion of attributes per object used earlier. Their experiments, as well as ours, show a quadratic growth of the number of concepts with increasing context size.

In [6] and [7] Godin et al. suspected the lattice size to increase linearly with the number of attributes per object. Our experiments suggest that this may be only the case when this number is small: looking at a fixed fill ratio the average number of attributes per object increases linearly with the context size $|\mathcal{I}|$. At least at a fill ration around 0.1 we found in Fig. 5 the lattice size to increase quadratically rather than linearly. A similar note was made by Carpineto et al. in [1].

6 Conclusions

Computing all concepts from a context $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ takes time linear to the size of the resulting concept lattice $L(\mathcal{G}, \mathcal{M}, \mathcal{I})$. This holds for Algorithm NextConcept that computes all concepts, as well as Algorithm Lattice that additionally computes the neighbors of all concepts. Since most applications require the lattice structure explicitly, and Algorithm Lattice is easy to implement, it is an all-purpose concept analysis algorithm.

Memory requirements and locality are two main differences between Lattice and NextConcept: the former needs random access to all concepts and thus requires them to be stored in main memoryr. The latter computes concepts one by one, so they can be written to a file. Using Algorithm Neighbors, which is part of Algorithm Lattice, just the neighborhood of

a concept can be determined without computing the rest of the lattice. This is difficult to achieve using Algorithm NextConcept since the total order that concepts are computed in does not respect neighborship.

Concept lattices can grow exponentially with respect to their underlying relation. However, at least when their context tables are sparsely filled, they tend to grow only quadratically where the density of context tables controls the growth rate. The modest growth found in experiments together with the presented efficient algorithm lead even overall to fast concept analysis.

References

[1] C. Carpineto and G. Romano. A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning*, 24:95–122, 1996.

[2] Thomas H. Cormen, Charles E. Leierson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, USA, 1990.

[3] Bernhard Ganter. *Beiträge zur Begriffsanalyse*, chapter Algorithmen zur Formalen Begriffsanalyse. BI-Wissenschaftsverlag, 1987.

[4] Bernhard Ganter and Sergei O. Kuznetsov. Stepwise construction of the Dedekind-MacNeille completion. In *Proc. 6th International Conference on Conceptual Structures*, Montpellier, August 1998.

[5] Bernhard Ganter and Rudolf Wille. *Formale Begriffsanalyse – Mathematische Grundlagen*. Springer, Berlin, 1996.

[6] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[7] Robert Godin, Rokia Missaoui, and Alain April. Experimental comparison of navigation in a galois lattice with conventional information rertrieval methods. *Int. J. Man-Machine Studies*, 38:747–767, 1993.

[8] Xavier Leroy. Objective Caml. Open-source licensed functional programming language. `http://caml.inria.fr/`.

[9] Christian Lindig. Concept-based component retrieval. In Jana Köhler, Fausto Giunchiglia, Cordell Green, and Christoph Walther, editors,

Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs, pages 21–25, Montreál, August 1995.

[10] Christian Lindig. Concepts. `ftp://ftp.ips.cs.tu-bs.de/pub/local/-softech/misc/concepts-0.3d`, 1997. Open Source implementation of concept analysis in C.

[11] Christian Lindig. *Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken.* PhD thesis, Technische Universität Braunschweig, D–38106 Braunschweig, Germany, November 1999. `http://www.eecs.harvard.edu/~lindig/papers/`.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[13] Dieter Schütt. Abschätzung f"ur die Anzahl der Begriffe von Kontexten. Master's thesis, Technische Hochschule Darmstadt, AG1: Allgemeine Algebra, Fachbereich Mathematik, Darmstadt, May 1987.

[14] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *Proc. of the Internation Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, October 1997.

[15] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proc. SIGSOFT Symposium on Foundations of Software Engineering.* ACM, 1998.

[16] Gerd Stumme and Rudolf Wille, editors. *Begriffliche Wissensverarbeitung – Methoden und Anwendungen.* Springer, 2000.

[17] F. Vogt and R. Wille. TOSCANA — a graphical tool for analyzing and exploring data. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 226–233. DIMACS, Springer-Verlag, October 1994. ISBN 3-540-58950-3.