

Formal Concept Analysis and Lexical Semantics

Jan van Eijck and Joost Zwarts

jve@cwi.nl, Joost.Zwarts@let.uu.nl

September 10, 2004

Abstract

To ascertain that a formalization of the intuitive notion of a ‘concept’ is linguistically interesting, one has to check whether it allows to get a grip on distinctions and notions from lexical semantics. Prime candidates are notions like ‘prototype’, ‘stereotypical attribute’, ‘essential attribute versus accidental attribute’, ‘intension versus extension’. We will argue that although the current paradigm of formal concept analysis as an application of lattice theory is not rich enough for an analysis of these notions, a lattice theoretical approach to concepts is a suitable starting point for formalizing them.

1 Introduction

Our goal in this paper is as follows. We will investigate the paradigm of formal concept analysis [2, Ch 3], [4] to see whether it allows us to get a grip on distinctions and notions from lexical semantics. We will focus on notions like ‘prototype’, ‘stereotypical attribute’, and distinctions like ‘essential attribute versus accidental attribute’, ‘intension versus extension’. The conclusion of our investigation will be that a lattice theoretical approach to concepts is a suitable starting point for formalizing such notions and distinctions.

Formal Concept Analysis (henceforth FCA) can be viewed as a formalization of what is sometimes called the classical theory of concepts [9]. In this theory a

concept is defined by a list of necessary and sufficient attributes. The attributes are necessary in the sense that an object can only fall under the concept if it has all the attributes of the concept. The list of attributes is sufficient to determine membership because every object that has these attributes falls under the concept. This is represented in the way FCA represents concepts as pairs of an extent (the set of objects that have all the attributes of the concept) and an intent (the set of attributes that are shared by all the objects falling under the concept). The classical theory accounts for the core properties of categorization in language, cognition and science: how the definitional structure of a concept determines its reference and its relations and combinations with other concepts. A Formal Concept Lattice demonstrates in a formal way how the concepts in a particular domain hang together through a partial ordering and the join and meet operations that combine concepts.

In spite of its theoretical appeal and prima facie empirical adequacy, the classical theory cannot give us a full account of categorization in language and cognition (as shown by many of the papers in [10]). This has become especially clear in the domain of lexical semantics where application of the classical theory has led to limited success. Although some words and lexical domains have been insightfully analyzed in terms of necessary and sufficient attributes (like bachelor and kinship terms), for many other areas an adequate definitional analysis along classical lines is hard to come up with. That something is either lacking or fundamentally wrong in the classical analysis has become evident through a series of influential works, starting with Wittgenstein's famous discussion of games [13], through Berlin and Kay's work on colour categorization [1] and Labov's work on vagueness [8], Putnam's philosophical arguments [11], Fillmore's lexical semantic explorations [3] and the typicality experiments of Rosch [12]. Categories are often not defined by one set of necessary and sufficient conditions, their boundaries may be vague and their membership graded. Members of a category may be more or less prototypical or stereotypical.

There is an ongoing debate on how the different facts about word meaning affect the classical theory of concepts. While some approaches have replaced the classical theory with a theory exclusively based on prototypes altogether, other approaches seek to combine the classical theory with components that can account for typicality effects. Our approach is in the latter tradition. Rather than trying to add to the philosophical debate, we have set ourselves the more modest task of exploring the usefulness of a suitably adapted version of FCA as a formal tool for analyzing lexical semantic phenomena.

2 Basic Notions

Let us start with a review of the theory. At the core of FCA is a basic distinction between *objects* and *attributes*. Objects are (thought of as) things in the world, attributes are properties that things in the world can have.

FCA represents a concept formally as a pair consisting of a set of objects X and a set of attributes Y , linked together by the following requirements:

1. X is the set of objects that satisfy all attributes in Y (every attribute in Y is shared by every object in X , no attribute outside Y is shared by every object in X),
2. Y is the set of attributes that specifies X (every object in X satisfies all attributes in Y , no object outside X has all attributes in Y).

Ingredients of the formalization of this are a domain O of objects, a domain A of attributes, and a relation H on $O \times A$, with $(o, a) \in H$ expressing that o has a .

A triple (O, A, H) is called a *context*. In a given context (O, A, H) , a pair (X, Y) with $X \subseteq O$ and $Y \subseteq A$ is a concept if

1. $X = \{x \in O \mid \forall y \in Y \ xHy\}$,
2. $Y = \{y \in A \mid \forall x \in X \ xHy\}$.

If (X, Y) is a concept, X is called its extent, Y its intent.

Example of a Context

The following zoological example context has ten objects and nine attributes.

	walks	quacks	lays eggs	feathered	warmblooded	flies	sings	small	suckles
robin			✓	✓	✓	✓	✓	✓	
dove			✓	✓	✓	✓		✓	
vulture			✓	✓	✓	✓			
ostrich	✓		✓	✓	✓				
bat					✓	✓		✓	✓
cow	✓				✓				✓
platypus	✓		✓		✓				✓
crocodile	✓		✓						
frog		✓	✓					✓	
butterfly			✓			✓		✓	

A platypus or duckbill is a small Australian animal which suckles its young but lays eggs. It is called duckbilled because it has a bill like that of a duck.

The resulting concepts are:

({ robin,dove,vulture,ostrich,bat,cow,platypus,crocodile,frog,butterfly },∅)
 ({ robin,dove,vulture,ostrich,bat,cow,platypus },{ warmblood })
 ({ robin,dove,vulture,ostrich,platypus,crocodile,frog,butterfly },{ eggs })
 ({ robin,dove,vulture,ostrich,platypus },{ eggs,warmblood })
 ({ robin,dove,vulture,ostrich },{ eggs,feathers,warmblood })
 ({ robin,dove,vulture,bat,butterfly },{ flies })
 ({ robin,dove,vulture,bat },{ warmblood,flies })
 ({ robin,dove,vulture,butterfly },{ eggs,flies })
 ({ robin,dove,vulture },{ eggs,feathers,warmblood,flies })
 ({ robin,dove,bat,frog,butterfly },{ small })
 ({ robin,dove,bat,butterfly },{ flies,small })
 ({ robin,dove,bat },{ warmblood,flies,small })
 ({ robin,dove,frog,butterfly },{ eggs,small })
 ({ robin,dove,butterfly },{ eggs,flies,small })
 ({ robin,dove },{ eggs,feathers,warmblood,flies,small })
 ({ robin },{ eggs,feathers,warmblood,flies,sings,small })
 ({ ostrich,cow,platypus,crocodile },{ walks })
 ({ ostrich,cow,platypus },{ walks,warmblood })
 ({ ostrich,platypus,crocodile },{ walks,eggs })
 ({ ostrich,platypus },{ walks,eggs,warmblood })
 ({ ostrich },{ walks,eggs,feathers,warmblood })
 ({ bat,cow,platypus },{ warmblood,suckles })
 ({ bat },{ warmblood,flies,small,suckles })
 ({ cow,platypus },{ walks,warmblood,suckles })
 ({ platypus },{ walks,eggs,warmblood,suckles })
 ({ frog },{ quacks,eggs,small })
 (∅, { walks,quacks,eggs,feathers,warmblood,flies,sings,small,suckles })

Ordered Sets

An ordered set is a pair (X, \leq) , where \leq is a binary relation on X that is

- reflexive: $\forall x \in X : x \leq x$,
- transitive: $\forall x, y, z \in X : ((x \leq y \wedge y \leq z) \rightarrow x \leq z)$,
- antisymmetric: $\forall x, y \in X : ((x \leq y \wedge y \leq x) \rightarrow x = y)$.

\leq is called a partial order on X or an order on X .

(Least) Upper Bounds, (Greatest) Lower Bounds

Let (X, \leq) be an ordered set.

If $x, y \in X$ then $\{x, y\}^u$, the set of upper bounds of $\{x, y\}$, is the set $\{z \in X \mid x \leq z, y \leq z\}$.

If $x, y \in X$ then $\{x, y\}^l$, the set of lower bounds of $\{x, y\}$, is the set $\{z \in X \mid z \leq x, z \leq y\}$.

If v is an upper bound of $\{x, y\}$, and moreover, $v \leq u$ for all upper bounds u of $\{x, y\}$, then v is a least upper bound of $\{x, y\}$. Notation: $\sup\{x, y\}$.

If w is a lower bound of $\{x, y\}$, and moreover, $w \geq l$ for all lower bounds l of $\{x, y\}$, then w is a greatest lower bound of $\{x, y\}$. Notation: $\inf\{x, y\}$.

Let (X, \leq) be an ordered set.

If $Y \subseteq X$ then Y^u , the set of upper bounds of Y , is the set

$$\{z \in X \mid \forall y \in Y : y \leq z\}.$$

If $Y \subseteq X$ then Y^l , the set of lower bounds of Y , is the set

$$\{z \in X \mid \forall y \in Y : z \leq y\}.$$

If v is an upper bound of Y , and moreover, $v \leq u$ for all upper bounds u of Y , then v is a least upper bound of Y . Notation: $\sup(Y)$

If w is a lower bound of Y , and moreover, $w \geq l$ for all lower bounds l of Y , then w is a greatest lower bound of Y . Notation: $\inf(Y)$.

Lattices, Complete Lattices

Let (X, \leq) be an ordered set.

If for every pair $x, y \in X$, $\sup\{x, y\}$ and $\inf\{x, y\}$ exist then (X, \leq) is called a lattice.

If (X, \leq) is a lattice, we write $\sup\{x, y\}$ as $x \vee y$, and $\inf\{x, y\}$ as $x \wedge y$.

$x \vee y$ is also called the *join* of x and y ; $x \wedge y$ is also called the *meet* of x and y .

If for every $Y \subseteq X$, $\sup(Y)$ and $\inf(Y)$ exist, then (X, \leq) is called a complete lattice.

If (X, \leq) is a complete lattice, we write $\sup(S)$ as $\bigvee S$ and $\inf(S)$ as $\bigwedge S$. Every finite lattice is complete. Infinite complete lattices have infinite joins and meets.

Top and Bottom Elements

If (X, \leq) is an ordered set, and there exists an element $x \in X$ with $y \leq x$ for all $y \in X$, then x is called the *top* element of X . Notation: \top .

If (X, \leq) is an ordered set, and there exists an element $x \in X$ with $x \leq y$ for all $y \in X$, then x is called the *bottom* element of X . Notation: \perp .

If (X, \leq) is a complete lattice, then $\sup(X) = \inf(\emptyset) = \top$ and $\inf(X) = \sup(\emptyset) = \perp$.

Formal Concept Lattices

The concept lattice that is determined by sets O and A , and relation $H \subseteq O \times A$ is the set (P, \leq) , where

$$P = \{(X, Y) \mid X \subseteq O, Y \subseteq A, (X, Y) \text{ is a concept}\},$$

and \leq is given by:

$$(X_1, Y_1) \leq (X_2, Y_2) := X_1 \subseteq X_2.$$

It follows immediately from this definition that $(X_1, Y_1) \leq (X_2, Y_2) \equiv Y_1 \supseteq Y_2$.

If $(X_1, Y_1) \leq (X_2, Y_2)$, then (X_1, Y_1) is called a subconcept of (X_2, Y_2) , and (X_2, Y_2) a superconcept of (X_1, Y_1) .

Recipes for Concept Construction

If $X \subseteq O$ then X' is given by $\{y \in A \mid \forall x \in X : xHy\}$.

This defines $' : \mathcal{P}(O) \rightarrow \mathcal{P}(A)$. Intuitively, X' denotes the set of those attributes that are shared by all members of X .

If $Y \subseteq A$ then Y' is given by $\{x \in O \mid \forall y \in Y : xHy\}$. This defines $' : \mathcal{P}(A) \rightarrow \mathcal{P}(O)$. Intuitively, Y' denotes the set of all objects that share the attributes in Y .

The notation $'$ is overloaded, but the argument always makes clear which function is meant.

It is easy to verify that (X, Y) is a concept iff $X = Y'$ and $Y = X'$ iff $X = X''$ and $Y = Y''$ iff $Y = Y''$ and $X = X''$. Thus, if $X \subseteq O$ then (X'', X') is a concept. Similarly, if $Y \subseteq A$ then (Y', Y'') is a concept.

Meets and Joins

The meet of two concepts (X_1, Y_1) and (X_2, Y_2) is given by

$$(X_1 \cap X_2, (Y_1 \cup Y_2)'').$$

It is easy to check that

$$(X_1 \cap X_2, (Y_1 \cup Y_2)'') = (X_1 \cap X_2, (X_1 \cap X_2)').$$

The join of two concepts (X_1, Y_1) and (X_2, Y_2) is given in dual fashion by

$$((X_1 \cup X_2)'', Y_1 \cap Y_2).$$

Again, it is easy to check that

$$((X_1 \cup X_2)'', Y_1 \cap Y_2) = ((Y_1 \cap Y_2)', Y_1 \cap Y_2).$$

In fact, concept lattices are *complete*, i.e., meets and joins of arbitrary sets (X_i, Y_i) of concepts exist. The meet of $\{(X_i, Y_i) \mid i \in I\}$ is given by

$$\left(\bigcap_{i \in I} X_i, \left(\bigcup_{i \in I} Y_i \right)'' \right),$$

the join by

$$\left(\left(\bigcup_{i \in I} X_i \right)'' , \bigcap_{i \in I} Y_i \right).$$

3 Extended Concepts

If the classical theory as it stands can not account for prototypes, then neither can FCA. There is nothing in the definition of a concept as a pair of objects and a pair of attributes that gives us any information about what its prototype might be or the prototypicality ratings of its members. What is needed is an extended notion of concept that allows for the representation of prototypical objects and stereotypical attributes.

Prototypical objects are objects that somehow exemplify a concept to a greater extent than non-prototypical objects. Similarly, stereotypical attributes are attributes that are somehow more basic than non-stereotypical attributes. We will now attempt to formalize these notions.

First we define extended contexts. An extended context is just like a context, except for the fact that it singles out a subset from the set of attributes as *essential* attributes. Formally, an extended context is a quadruple (O, A, H, E) such that (O, A, H) is a context, and $E \subseteq A$.

Let an extended context (O, A, H, E) be given. Let (X, P, Y, Q) be a quadruple with $P \subseteq X \subseteq O$ and $Y \subseteq Q \subseteq A$. We say that (X, P, Y, Q) is an extended concept in (O, A, H, E) when (X, Y) and (P, Q) are both concepts in (O, A, H) , and moreover $Y \subseteq E$ (all attributes that every object in the concept has are essential attributes).

Note that it follows immediately from this definition that $(P, Q) \leq (X, Y)$, in other words, that (X, Y) is a super-concept of (P, Q) . We have a borderline case where $X = P$ and $Y = Q$ (an ordinary concept viewed as an extended concept).

If (X, P, Y, Q) is an extended concept, then X is its extent, Y its intent, P are its prototypes, $Q - Y$ are its stereotypical attributes.

The stereotypical attributes are the attributes that all prototypes share, but that non-prototypical objects falling under the concept may lack. Indeed, every non-prototypical object falling under the concept will lack at least one of the

stereotypical attributes.

Being Closer to the Prototypical Case...

We can give a relation for 'being closer to the prototypical case' on the extent of an extended concept (X, P, Y, Q) by defining \preceq as follows:

$$x_1 \preceq x_2 := \forall y \in (Q - Y)(x_2Hy \rightarrow x_1Hy).$$

Intuitively, $x_1 \preceq x_2$ expresses that x_1 has all the stereotypical attributes of x_2 .

This defines a pre-order relation on X , i.e., \preceq is both reflexive and transitive.

If we look at the so-called poset reflection of \preceq , by identifying objects x_1, x_2 in case $x_1 \preceq x_2$ and $x_2 \preceq x_1$, we see that all the prototypes get identified, all the elements lacking all the attributes in $Q - Y$ get identified, and more generally, all objects lacking a particular set A_1 of stereotypical attributes are in the same class.

The smallest equivalence class (in the ordering \preceq) is the set P , the largest equivalence class (in the ordering \preceq) is the set of objects in X lacking all the stereotypical attributes. This is the following set:

$$\{x \in X \mid \forall y \in (Q - Y) \neg xHy\}.$$

The Lattice of Extended Concepts

Extended concepts again form a complete lattice. Meets are now given by

$$\left(\bigcap_{i \in I} X_i, \bigcap_{i \in I} P_i, \left(\bigcup_{i \in I} Y_i \right)'' , \left(\bigcup_{i \in I} Q_i \right)'' \right),$$

joins by

$$\left(\left(\bigcup_{i \in I} X_i \right)'' , \left(\bigcup_{i \in I} P_i \right)'' , \bigcap_{i \in I} Y_i, \bigcap_{i \in I} Q_i \right).$$

Thus, the present formalization would predict that the prototypical cases of the concept that results from combining (X_1, P_1, Y_1, Q_1) and (X_2, P_2, Y_2, Q_2) are the objects in $P_1 \cap P_2$.

Prototypes as a Limited Form of Intensionality

The notion of a concept in formal concept analysis is completely extensional. If (X_1, Y_1) and (X_2, Y_2) are concepts with $X_1 = X_2$ then the concepts are the same. A concept is fully determined by its extent.

For extended concepts this is no longer so. If (X_1, P_1, Y_1, Q_1) and (X_2, P_2, Y_2, Q_2) are extended concepts and their extents are the same, their sets of prototypes may still vary. The extent of an extended concept does no longer determine the concept, simply because an extended concept is more than just a pair consisting of a extent and an intent that mutually determine one another.

Example Extended Context

To make the zoological example context of page 3 into an extended context, we must specify its essential attributes. It is reasonable to single out *egg-laying* and *warmblooded* as essential, to express that we want to focus on birds. (In a different example, singling out *warmblooded*, and *suckling* as essential would provide an appropriate way for focussing at mammals.)

Here are some non-trivial examples of extended concepts for this extended context:

{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{robin,dove,vulture,ostrich,platypus}, {eggs,warmblood})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{robin,dove,vulture,ostrich}, {eggs,feathers,warmblood})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{robin,dove,vulture,bat}, {warmblood,flies})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{robin,dove,vulture}, {eggs,feathers,warmblood,flies})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{robin,dove,bat}, {warmblood,flies,small})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{robin,dove}, {eggs,feathers,warmblood,flies,small})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{robin}, {eggs,feathers,warmblood,flies,sings,small})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{ostrich,cow,platypus}, {walks,warmblood})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{ostrich,platypus}, {walks,eggs,warmblood})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{ostrich}, {walks,eggs,feathers,warmblood})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{bat,cow,platypus}, {warmblood,suckles})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{bat}, {warmblood,flies,small,suckles})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{cow,platypus}, {walks,warmblood,suckles})
{robin,dove,vulture,ostrich,bat,cow,platypus}, {warmblood},	{platypus}, {walks,eggs,warmblood,suckles})
{bat,cow,platypus}, {warmblood,suckles},	{bat}, {warmblood,flies,small,suckles})
{bat,cow,platypus}, {warmblood,suckles},	{cow,platypus}, {walks,warmblood,suckles})
{bat,cow,platypus}, {warmblood,suckles},	{platypus}, {walks,eggs,warmblood,suckles})

Take the example extended context with

$$\{\text{robin,dove,vulture,ostrich,bat,cow,platypus}\}$$

as the extent, $\{\text{robin,dove}\}$ as the prototypes, $\{\text{warmblood}\}$ as the essential attribute, and $\{\text{eggs,feathers,flies,small}\}$ as the stereotypical attributes. This puts the birds and the mammals together, and it singles out the small feathered flying egg layers as prototypes.

If we allow the extent and the prototypes to coincide then we still do not quite get out the concept of a bird, due to the presence of the duckbill: we get $\{\text{robin,dove,vulture,ostrich,platypus}\}$ as extent, the same objects as essential objects, and the two essential attributes as extent.

One way to get rid of the platypus, is by calling in the stereotypical attribute *feathered*. This gives the birds as prototypical examples. with the attributes *egg-laying*, *warmblooded*, *feathered*.

4 Intensional Concepts

Let again a set O of objects and a set A of attributes be given.

For an intensional analysis of concepts we introduce a set of possible states of affairs or possible worlds W , and we let H be a function from W to $\mathcal{P}(O \times A)$. Intuitively, H maps each possible world w to a relation H_w that determines which objects have which attributes in world w .

An intensional concept has an extent and an intent in every world: it is a pair of functions $F : W \rightarrow \mathcal{P}(O)$, $G : W \rightarrow \mathcal{P}(A)$ with the property that for every $w \in W$, the pair $(F(w), G(w))$ is a concept.

An intensional concept is a pair (F, G) with the following properties.

1. $F : W \rightarrow \mathcal{P}(O)$ (F is a function from worlds to sets of objects),
2. $G : W \rightarrow \mathcal{P}(A)$ (G is a function from worlds to sets of attributes),
3. $\forall w \in W : F'(w) = G(w)$, where F' is the function in $W \rightarrow \mathcal{P}(A)$ given by

$$F'(w) := \{a \in A \mid \forall o \in F(w) \ o H_w a\},$$

4. $\forall w \in W : G'(w) = F(w)$, where G' is the function in $W \rightarrow \mathcal{P}(O)$ given by

$$G'(w) := \{o \in O \mid \forall a \in G(w) \ oH_w a\}.$$

Essential and Accidental Attributes

In an intensional concept, the essential attributes are the attributes in the set $\bigcap_{w \in W} G(w)$: those attributes that belong to the intent of the concept in every world. Presumably, concepts for natural kinds have such essential attributes, while concepts for artificial kinds need not have them.

Intensionality gives us a handle on the distinction between essential and accidental attributes, but by itself does not give us prototypes and stereotypes. One way to get prototypes back is by singling out one world w_p as a prototype world...

Extended Intensional Concepts

An extended intensional concept is a triple (F, G, w_p) such that (F, G) is an intensional concept, and w_p is a member of W with the property that for all $w \in W$, $F(w_p) \subseteq F(w)$. This expresses that in every world, the prototypical objects are in the extent of the concept. From $F(w_p) \subseteq F(w)$ it follows that $G(w) \subseteq G(w_p)$. Since w is arbitrary, we have $\bigcap_{w \in W} G(w) \subseteq G(w_p)$, and we see that the prototypical objects share all the essential properties of an intensional concept.

5 Conclusions

Although Formal Concepts in the sense of FCA by themselves are not rich enough to model interesting notions and distinctions from lexical semantics, FCA still is an interesting point of departure for developing (a formal version of) lexical semantics. In the paper we hope to have shown the promise of an approach that uses the apparatus of FCA as a point of departure for the development of gradual enrichments that do more justice to linguistic insights, while remaining formally manageable.

In future work we hope to explore this road further. Once one gets serious about

putting lexical semantics on a formal footing, there are lots and lots of work on the agenda. It would be interesting to extend the account to lexical negation, formally distinguishing between the various kinds from the semantic taxonomy. Also, connections between different contexts should be explored, and finally, the whole apparatus should be set to work in the analysis of specific lexical domains.

References

- [1] B. Berlin and P. Kay. *Basic Color Terms: Their Universality and Evolution*. University of California Press, Berkeley, 1969.
- [2] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order (Second Edition)*. Cambridge University Press, Cambridge, 2002. First edition: 1990.
- [3] J.C. Fillmore. An alternative to checklist theories of meaning. In C. Cogen et al., editor, *Proceedings of the First Annual Meeting of the Berkeley Linguistic Society*, pages 123–131, 1975.
- [4] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer, 1999.
- [5] The Haskell Team. The Haskell homepage. <http://www.haskell.org>.
- [6] S. Peyton Jones, editor. *Haskell 98 Language and Libraries; The Revised Report*. Cambridge University Press, 2003.
- [7] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.
- [8] W. Labov. The boundaries of words and their meanings. In C.J. Bailey and R. Shy, editors, *New Ways of Analyzing Variation in English*, pages 340–373. Georgetown University Press, Washington, 1973.
- [9] S. Laurence and E. Margolis. Concepts and cognitive science. In E. Margolis and S. Laurenc, editors, *Concepts: Core Readings*, pages 3–81. MIT Press/Bradford Books, 1999.
- [10] E. Margolis and S. Laurence, editors. *Concepts: Core Readings*. MIT Press/Bradford Books, 1999.

- [11] H. Putnam. Is semantics possible? In H. Kiefer and M. Munitz, editors, *Language, Belief and Metaphysics*, pages 50–63. State University of New York Press, New York, 1970.
- [12] E. Rosch. On the internal structure of perceptual and semantic categories. In T. Moore, editor, *Cognitive Development and the Acquisition of Language*, pages 111–144. Academic Press, New York, 1973.
- [13] L. Wittgenstein. *Philosophical Investigations*. Blackwell, 1953.

Appendix: Implementation

This appendix gives an implementation in Haskell [5, 6], in ‘literate programming’ style [7], of the notions introduced in this paper.

Contexts and Concepts

```
module FCA where

import List
import Char
```

A context is a triple consisting of a list of objects (type a), a list of attributes (type b), and a relation between objects and attributes (represented as a function $a \rightarrow b \rightarrow \text{Bool}$).

```
type Cntxt a b = ([a],[b], a -> b -> Bool)
```

The function $' :: \mathcal{P}(O) \rightarrow \mathcal{P}(A)$. The function has a context as parameter.

```
oprime :: (Ord a, Ord b) => Cntxt a b -> [a] -> [b]
oprime (objects,attributes,has) xs =
  [ y | y <- attributes, all (\x -> has x y) xs ]
```

The function $' :: \mathcal{P}(A) \rightarrow \mathcal{P}(O)$. The function has a context as parameter.

```

aprime :: (Ord a, Ord b) => Cntxt a b -> [b] -> [a]
aprime (objects,attributes,has) ys =
  [ x | x <- objects, all (\y -> has x y) ys ]

```

Check whether (X, Y) is a concept in a given context.

```

isConcept :: (Ord a, Ord b) =>
  Cntxt a b -> ([a],[b]) -> Bool
isConcept context (xs,ys) =
  oprime context xs == ys
  &&
  aprime context ys == xs

```

The powerlist function.

```

powerList :: [a] -> [[a]]
powerList [] = [[]]
powerList (x:xs) =
  (map (x:) (powerList xs)) ++ (powerList xs)

```

List all concepts of a given context.

```

concepts :: (Ord a, Ord b) =>
  Cntxt a b -> [[a],[b]]
concepts context@(objects,attributes,has) =
  [ (xs,ys) | xs <- powerList objects,
    ys <- [ oprime context xs ],
    aprime context ys == xs ]

```

Give just the number of concepts of a given context:

```
nconcepts :: (Ord a, Ord b) => Cntxt a b -> Int
nconcepts = length . concepts
```

The following function displays lists in a convenient way, with every item on a separate line.

```
display :: Show a => [a] -> IO()
display [] = return ()
display (x:xs) = do print x
                    display xs
```

Examples

Example 1

```
objects1    = ['a'..'f']

attributes1 = ['0'..'2']

has1 :: Char -> Char -> Bool
has1 x '0' = elem x "ae"
has1 x '1' = notElem x "ae"
has1 x '2' = elem x "abcd"

context1 = (objects1,attributes1,has1)
```

This context has 7 concepts:

```
FCA> display (concepts context1)
("abcdef", "")
("abcd", "2")
("ae", "0")
("a", "02")
("bcdf", "1")
("bcd", "12")
("", "012")
```

Exercise 1 *How can one change this context is such a way that one gets eight concepts? Note that a concept with attributes 01 is missing.*

```
FCA> aprime context1 "01"
""
FCA> oprime context1 ""
"012"
```

Example 2

```
objects2    = ['a'..'f']

attributes2 = ['0'..'2']

has2 :: Char -> Char -> Bool
has2 x '0' = elem x "aef"
has2 x '1' = notElem x "ae"
has2 x '2' = elem x "abcd"

context2 = (objects2, attributes2, has2)
```

```
FCA> nconcepts context2
8
FCA> display (concepts context2)
("abcdef", "")
```

```
("abcd", "2")
("aef", "0")
("a", "02")
("bcdf", "1")
("bcd", "12")
("f", "01")
("", "012")
```

Example 3

```
objects3    = ['a'..'j']

attributes3 = ['0'..'3']

has3 :: Char -> Char -> Bool
has3 x '0' = elem x "aei"
has3 x '1' = notElem x "ae"
has3 x '2' = elem x "abcd"
has3 x '3' = elem x "defgh"

context3 = (objects3, attributes3, has3)
```

This context has 12 concepts:

```
FCA> nconcepts context3
12
FCA> display (concepts context3)
("abcdefghij", "")
("abcd", "2")
("aei", "0")
("a", "02")
("bcdfghij", "1")
("bcd", "12")
("defgh", "3")
("dfgh", "13")
```

```
("d", "123")
("e", "03")
("i", "01")
("", "0123")
```

Example 4

```
objects4    = ['a'..'j']
attributes4  = ['0'..'3']

has4 :: Char -> Char -> Bool
has4 x '0' = elem x "a"
has4 x '1' = elem x "bcd"
has4 x '2' = elem x "aef"
has4 x '3' = elem x "ghi"

context4 = (objects4, attributes4, has4)
```

This context has 6 concepts:

```
FCA> nconcepts context4
6
FCA> display (concepts context4)
("abcdefghij", "")
("aef", "2")
("a", "02")
("bcd", "1")
("ghi", "3")
("", "0123")
```

Exercise 2 Give a context with 10 objects and 4 attributes that has 16 concepts.

Example 5

Conversion function for relations represented as functions of type $a \rightarrow [b]$.

```
conv :: Eq b => (a -> [b]) -> a -> b -> Bool
conv r x y = elem y (r x)
```

This is used in the following example.

```
objects5    = ['a'..'h']
attributes5 = ['1'..'9']

has5 :: Char -> Char -> Bool
has5 = conv has where
  has :: Char -> [Char]
  has 'a' = "129" ; has 'b' = "159"
  has 'c' = "23"  ; has 'd' = "128"
  has 'e' = "268" ; has 'f' = "16"
  has 'g' = "6789" ; has 'h' = "45"

context5 = (objects5,attributes5,has5)
```

```
FCA> nconcepts context5
20
FCA> display (concepts context5)
("abcdefgh", "")
("abdf", "1")
("abg", "9")
("ab", "19")
("acde", "2")
("ad", "12")
("a", "129")
("bh", "5")
("b", "159")
("c", "23")
("deg", "8")
("de", "28")
```

```
("d", "128")
("efg", "6")
("eg", "68")
("e", "268")
("f", "16")
("g", "6789")
("h", "45")
("", "123456789")
```

Example 6

The zoological example from the main text.

```
objects6 =
  ["robin", "dove", "vulture", "ostrich", "bat",
   "cow", "platypus", "crocodile", "frog", "butterfly"]
attributes6 =
  ["walks", "quacks", "eggs", "feathers", "warmblood",
   "flies", "sings", "small", "suckles"]
has6 = conv has where
  has "robin"      = attributes6 \\ ["walks", "quacks", "suckles"]
  has "dove"       = attributes6 \\
                    ["walks", "quacks", "sings", "suckles"]
  has "vulture"   = ["eggs", "feathers", "warmblood", "flies"]
  has "ostrich"   = ["walks", "eggs", "feathers", "warmblood"]
  has "bat"       = ["warmblood", "flies", "small", "suckles"]
  has "cow"       = ["walks", "warmblood", "suckles"]
  has "platypus"  = ["walks", "eggs", "warmblood", "suckles"]
  has "crocodile" = ["walks", "eggs"]
  has "frog"      = ["quacks", "eggs", "small"]
  has "butterfly" = ["eggs", "flies", "small"]

context6 = (objects6, attributes6, has6)
```

Here are the resulting concepts:

```
FCA> display (concepts context6)
```

```

(["robin","dove","vulture","ostrich","bat","cow","platypus","crocodile","frog","butterfly"],[])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],["warmblood"])
(["robin","dove","vulture","ostrich","platypus","crocodile","frog","butterfly"],["eggs"])
(["robin","dove","vulture","ostrich","platypus"],["eggs","warmblood"])
(["robin","dove","vulture","ostrich"],["eggs","feathers","warmblood"])
(["robin","dove","vulture","bat","butterfly"],["flies"])
(["robin","dove","vulture","bat"],["warmblood","flies"])
(["robin","dove","vulture","butterfly"],["eggs","flies"])
(["robin","dove","vulture"],["eggs","feathers","warmblood","flies"])
(["robin","dove","bat","frog","butterfly"],["small"])
(["robin","dove","bat","butterfly"],["flies","small"])
(["robin","dove","bat"],["warmblood","flies","small"])
(["robin","dove","frog","butterfly"],["eggs","small"])
(["robin","dove","butterfly"],["eggs","flies","small"])
(["robin","dove"],["eggs","feathers","warmblood","flies","small"])
(["robin"],["eggs","feathers","warmblood","flies","sings","small"])
(["ostrich","cow","platypus","crocodile"],["walks"])
(["ostrich","cow","platypus"],["walks","warmblood"])
(["ostrich","platypus","crocodile"],["walks","eggs"])
(["ostrich","platypus"],["walks","eggs","warmblood"])
(["ostrich"],["walks","eggs","feathers","warmblood"])
(["bat","cow","platypus"],["warmblood","suckles"])
(["bat"],["warmblood","flies","small","suckles"])
(["cow","platypus"],["walks","warmblood","suckles"])
(["platypus"],["walks","eggs","warmblood","suckles"])
(["frog"],["quacks","eggs","small"])
([],["walks","quacks","eggs","feathers","warmblood","flies","sings","small","suckles"])

```

The 'covers' relation

If P is an ordered set with $x, y \in P$, then y covers x ($y \triangleright x$) or x is covered by y ($x \triangleleft y$), if $x < y$ and $x \leq z < y$ implies $x = y$.

To implement this for the special case of concepts we need implementations of \subseteq and \subsetneq :

```

containedIn :: Eq a => [a] -> [a] -> Bool
containedIn [] ys = True
containedIn (x:xs) ys =
    elem x ys && containedIn xs ys

pcontainedIn :: (Eq a,Ord a) => [a] -> [a] -> Bool
pcontainedIn xs ys = containedIn xs ys && xs' /= ys'
    where xs' = (sort.nub) xs
          ys' = (sort.nub) ys

```

Here is the covering relation for concepts:

```

covers :: Ord b => [[(a],[b])] -> ((a],[b]) -> ((a],[b]) -> Bool
covers concepts (xs,ys) (us,vs) =
    pcontainedIn ys vs && null subvs
    where
        attrlist = map snd concepts
        attrlist' = filter (/= ys) attrlist
        subvs     = filter
            (\ls -> containedIn ys ls && pcontainedIn ls vs)
            attrlist'

```

Break up a list of concepts into a list of its covering pairs:

```

covering :: Ord b => [[(a],[b])] -> [((a],[b]),((a],[b]))]
covering [] = []
covering (concept: concepts) =
    [ (concept,c) | c <- concepts,
        covers concepts concept c ]
    ++
    covering concepts

```

Graphical Display

The following function uses a string as glue between a list of strings, to produce a single long string.

```
glueWith :: String -> [String] -> String
glueWith _ []      = []
glueWith _ [y]     = y
glueWith s (y:ys) = y ++ s ++ glueWith s ys
```

Showing a link:

```
showLink :: Show a => (([a],[b]),([a],[b])) -> String
showLink ((xs,ys),(us,vs)) = source ++ " -> " ++ target
  where
    source = filter isAlpha (show xs)
    target' = filter isAlpha (show us)
    target  = if null target' then "\"\"" else target'
```

Mapping a covering to a graphics string:

```
graphviz :: Show a => [(([a],[b]),([a],[b]))] -> String
graphviz links =
  "digraph G { "
  ++
  glueWith " ; " [ showLink l | l <- links ]
  ++ " }"
```

Write graph to file:

```
writeGraph :: String -> IO()
writeGraph cts = writeFile "graph.dot" cts
```

```
writeGr :: String -> String -> IO()
writeGr name cts = writeFile name cts
```

Write lattice to file:

```
writeLattice :: (Show a, Show b, Ord a, Ord b) => String -> Cntxt a b -> IO()
writeLattice name context =
  writeGr (name ++ ".dot") (graphviz (covering (concepts context)))
```

Joins and Meets

Concepts over a context form a lattice, i.e., joins and meets are defined. The following function gives a unit list with the join of two pairs provided these pairs both are concepts in the given context (otherwise the empty list is returned).

```

join :: (Ord a, Ord b) => Cntxt a b ->
      ([a],[b]) -> ([a],[b]) -> [[a],[b]]
join context c1@(xs1,ys1) c2@(xs2,ys2)
  | not (isConcept context c1) = []
  | not (isConcept context c2) = []
  | otherwise                  =
    [ (xs3,ys3) | xs3 <- [ aprime context
                          (oprime context
                           (union xs1 xs2)) ],
      ys3 <- [ intersect ys1 ys2 ],
      isConcept context (xs3,ys3)    ]

```

Some examples:

```

FCA> join context1 ("bcdf","1") ("abcd","2")
[("abcdef","")]
FCA> join context1 ("bcd","1") ("abcd","2")
[]
FCA> join context1 ("","012") ("abcd","2")
[("abcd","2")]

```

The following function gives a unit list with the meet of two pairs provided these pairs both are concepts in the given context (otherwise the empty list is returned).

```

meet :: (Ord a, Ord b) => Cntxt a b ->
      ([a],[b]) -> ([a],[b]) -> [[([a],[b])]
meet context c1@(xs1,ys1) c2@(xs2,ys2)
  | not (isConcept context c1) = []
  | not (isConcept context c2) = []
  | otherwise                  =
    [ (xs3,ys3) | xs3 <- [ intersect xs1 xs2 ],
      ys3 <- [ oprime context
               (aprime context
                (union ys1 ys2))] ,
      isConcept context (xs3,ys3)      ]

```

Some examples:

```

FCA> meet context1 ("","012") ("abcd","2")
[("","012")]
FCA> meet context1 ("bcdf","1") ("abc2d","2")
[("bcd","12")]
FCA> meet context1 ("bcd","1") ("abcd","2")
[]

```

Extended Concepts

Here is a type for extended contexts. The final component is for the essential attributes.

```

type Xcntxt a b = ([a],[b], a -> b -> Bool,[b])

```

Extracting the context from an extended context is just a matter of deleting the fourth component:

```
x2c :: Xcntxt a b -> Cntxt a b
x2c (o,a,h,e) = (o,a,h)
```

Extracting the essential attributes from an extended context:

```
essence :: Xcntxt a b -> [b]
essence (o,a,h,e) = e
```

Check whether a given quadruple is an extended concept in a given extended context:

```
isXconcept :: (Ord a, Ord b) =>
             Xcntxt a b -> ([a],[a],[b],[b]) -> Bool
isXconcept xcontext (xs,ps,ys,qs) =
  containedIn ps xs
  &&
  containedIn ys qs
  &&
  isConcept (x2c xcontext) (xs,ys)
  &&
  isConcept (x2c xcontext) (ps,qs)
  &&
  containedIn ys (essence xcontext)
```

List alle extended concepts of a given extended context:

```

xconcepts :: (Ord a, Ord b) =>
  Xcntxt a b -> ([[a],[a],[b],[b]])
xconcepts xcontext@(objects,attributes,has,eattributes) =
  [ (xs,ps,ys,qs) | xs <- powerList objects,
    ps <- powerList xs,
    ys <- [ oprime (x2c xcontext) xs ],
    containedIn ys eattributes,
    aprime (x2c xcontext) ys == xs,
    qs <- [ oprime (x2c xcontext) ps ],
    aprime (x2c xcontext) qs == ps ]

```

Triviality test for extended concepts. Call an extended concept trivial if either all its objects are prototypes, or there are no prototypes at all, or there are no attributes at all.

```

trivial :: (Ord a, Ord b) => ([a],[a],[b],[b]) -> Bool
trivial (xs,ps,ys,qs) = xs == ps || ps == [] || ys == []

```

Prototypes and Stereotypes of an extended concept:

```

prototypes :: ([a],[a],[b],[b]) -> [a]
prototypes (xs,ps,ys,qs) = ps

stereotypes :: (Eq b) => ([a],[a],[b],[b]) -> [b]
stereotypes (xs,ps,ys,qs) = qs \\ ys

```

The 'covers' relation for extended concepts

```
xcovers :: (Eq a,Eq b) => [[a],[a],[b],[b]] ->
    ([a],[a],[b],[b]) -> ([a],[a],[b],[b]) -> Bool
xcovers xconcepts (xs,ps,ys,qs) (us,ws,vs,zs) =
    xs == us && covers (map (\(_,x,_,y) -> (x,y)) xconcepts) (ps,qs) (vs,zs)
    ||
    null ps && covers (map (\(x,_,y,_) -> (x,y)) xconcepts) (xs,ys)
```

```
xcovers :: Ord b => [[a],[a],[b],[b]]
    -> ([a],[a],[b],[b]) -> ([a],[a],[b],[b]) -> Bool
xcovers xconcepts (xs,ps,ys,qs) (us,ws,vs,zs) =
    pcontainedIn ys vs && null subvs && (covers (ps,qs) (ws,zs) || (null ps) &&
    where
        attrlist = map (\ (_,_,x,_) -> x) xconcepts
        attrlist' = filter (/= ys) attrlist
        subvs     = filter
            (\ls -> containedIn ys ls && pcontainedIn ls vs)
            attrlist'
```

Break up a list of extended concepts into a list of its covering pairs:

```
xcovering :: Ord b => [[a],[a],[b],[b]]
    -> [[([a],[a],[b],[b]),([a],[a],[b],[b])]]
xccovering [] = []
xccovering (xconcept: xconcepts) =
    [ (xconcept,c) | c <- xconcepts,
        xcovers xconcepts xconcept c ]
    ++
xccovering xconcepts
```

Display of Extended Concepts

Showing a link of an extended concept:

```
showXlink :: Show a => (([a],[a],[b],[b]),([a],[a],[b],[b])) -> String
showXlink ((xs,ps,ys,qs),(us,ws,vs,zs)) = source ++ " -> " ++ target
  where
    source = filter isAlpha (show xs) ++ "_" ++ filter isAlpha (show ps)
    target' = filter isAlpha (show us) ++ "_" ++ filter isAlpha (show ws)
    target = if null target' then "\\\" else target'
```

Mapping a covering of an extended concept lattice to a graphics string:

```
graphvizX :: Show a => [(([a],[a],[b],[b]),([a],[a],[b],[b]))] -> String
graphvizX links =
  "digraph G { "
  ++
  glueWith " ; " [ showXlink l | l <- links ]
  ++ " }"
```

Write extended concept lattice to file:

```
writeX :: (Show a, Show b, Ord a, Ord b) => String -> Xcntxt a b -> IO()
writeX name xcontext =
  writeGr (name ++ ".dot") (graphvizX (xcovering (xconcepts xcontext)))
```

Examples

```
xcontext0 = (o,a,h,ess)
  where
    (o,a,h) = context1
    ess = ['0']
```

```
xcontext1 = (o,a,h,ess)
  where
    (o,a,h) = context1
    ess = ['1']
```

Example of an extended context, based on the zoological example from the main text (example 6).

```
xcontext = (o,a,h,ess)
  where
    (o,a,h) = context6
    ess = ["eggs","warmblood"]
```

Check of 'isXconcept'

```
FCA> map (isXconcept xcontext) (xconcepts xcontext)
[True,True,True,True,True,True,True,True,True,True,True,True,
 True,True,True,True,True,True]
```

(Nontrivial) extended concepts of example context 6

```

FCA> display (filter (not . trivial) (xconcepts xcontext))
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["robin","dove","vulture","ostrich","platypus"],
 ["warmblood"],
 ["eggs","warmblood"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["robin","dove","vulture","ostrich"],
 ["warmblood"],
 ["eggs","feathers","warmblood"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["robin","dove","vulture","bat"],
 ["warmblood"],
 ["warmblood","flies"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["robin","dove","vulture"],
 ["warmblood"],
 ["eggs","feathers","warmblood","flies"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["robin","dove","bat"],
 ["warmblood"],
 ["warmblood","flies","small"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["robin","dove"],
 ["warmblood"],
 ["eggs","feathers","warmblood","flies","small"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["robin"],
 ["warmblood"],
 ["eggs","feathers","warmblood","flies","sings","small"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["ostrich","cow","platypus"],
 ["warmblood"],
 ["walks","warmblood"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
 ["ostrich","platypus"],
 ["warmblood"],
 ["walks","eggs","warmblood"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],

```

```

["ostrich"],
["warmblood"],
["walks","eggs","feathers","warmblood"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
["bat","cow","platypus"],
["warmblood"],
["warmblood","suckles"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
["bat"],
["warmblood"],
["warmblood","flies","small","suckles"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
["cow","platypus"],
["warmblood"],
["walks","warmblood","suckles"])
(["robin","dove","vulture","ostrich","bat","cow","platypus"],
["platypus"],["warmblood"],
["walks","eggs","warmblood","suckles"])
(["bat","cow","platypus"],["bat"],
["warmblood","suckles"],
["warmblood","flies","small","suckles"])
(["bat","cow","platypus"],
["cow","platypus"],
["warmblood","suckles"],
["walks","warmblood","suckles"])
(["bat","cow","platypus"],
["platypus"],["warmblood","suckles"],
["walks","eggs","warmblood","suckles"])

```

The stereotype attributes of an extended concept:

```

FCA> stereotypes
(["robin","dove","vulture","ostrich","crocodile"],
["robin"],
["eggs"],
["eggs","feathers","warmblood","flies","sings","small"])
["feathers","warmblood","flies","sings","small"]

```

Joins of extended concepts can be implemented in terms of joins of simple concepts:

```
xjoin :: (Ord a, Ord b) =>
  Xcntxt a b -> ([a],[a],[b],[b]) ->
    ([a],[a],[b],[b]) -> [[([a],[a],[b],[b])]
xjoin xcontext (xs1,ps1,ys1,qs1) (xs2,ps2,ys2,qs2) =
  [(xs,ps,ys,qs) |
    (xs,ys) <- join (x2c xcontext) (xs1,ys1) (xs2,ys2),
    (ps,qs) <- join (x2c xcontext) (ps1,qs1) (ps2,qs2) ]
```

Similarly for meets of extended contexts:

```
xmeet :: (Ord a, Ord b) =>
  Xcntxt a b -> ([a],[a],[b],[b]) ->
    ([a],[a],[b],[b]) -> [[([a],[a],[b],[b])]
xmeet xcontext (xs1,ps1,ys1,qs1) (xs2,ps2,ys2,qs2) =
  [(xs,ps,ys,qs) |
    (xs,ys) <- meet (x2c xcontext) (xs1,ys1) (xs2,ys2),
    (ps,qs) <- meet (x2c xcontext) (ps1,qs1) (ps2,qs2) ]
```