A simple concept learner and its implementation

Flach, P.A.

Publication date:
1990

Document Version
Publisher's PDF, also known as Version of record

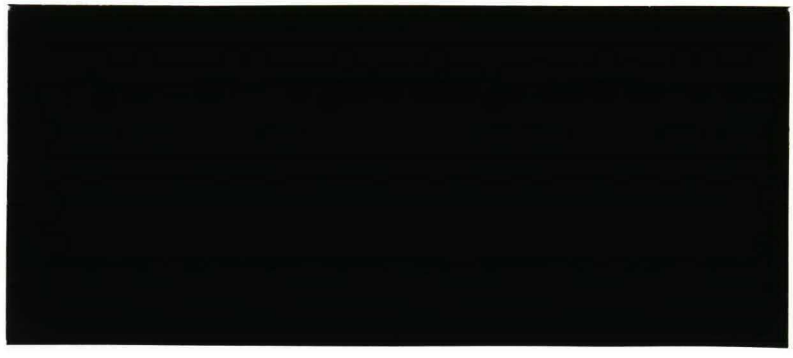
[redacted]

ITK

RESEARCH
REPORT

INSTITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE

INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE



ITK Research Report No. 9

January 1990

A simple concept learner and its implementation

Peter A. Flach

Revised version of a paper appearing in

Computing Science in the Netherlands CSN'89
P.M.G. Apers, D. Bosman, J. van Leeuwen (eds.),
Stichting Mathematisch Centrum, Amsterdam, 1989, pp. 149-171.

ISSN 0924-7807

Institute for Language Technology and Artificial Intelligence,
Tilburg University, The Netherlands

A simple concept learner and its implementation

Peter A. Flach

ABSTRACT

An implementation of a simple concept learner is developed, using Prolog's unification mechanism. A formal, algebraic analysis of the syntax and semantics of the representation formalism used precedes and aids in the derivation of the program. The program can generate various forms of hypotheses (i.e., most specific, more general, and intermediate), depending on the interpretation of negative examples.

Contents

1. Introduction

1.1 Purpose and scope

Machine learning is an important branch of Artifical Intelligence. A form of machine learning especially suited for formalisation and automation is inductive learning or *learning from examples*. When examples are instances and non-instances of a concept to be discovered, we speak of *concept learning*. This paper is devoted to the subject of concept learning from examples. The purpose of this paper is twofold: first, we attempt a deeper analysis of the concept learning problem. Second, we develop an implementation for a specific concept learning problem. These goals are closely related: it is our conviction, that a formal analysis aids in the development of an implementation. Conversely, we are aware of the fact that abstract notions often need clarification, which may be provided by examples.

1.2 Intensions, extensions, and representations

A distinction is usually drawn between the *intension* and the *extension* of a concept. The extension of a concept is the set of things associated with it; the intension of a concept is its intended meaning, which is not solely dependent on whether it corresponds to any existing thing in the real world. For instance, as far as we know, there do not exist unicorns, hence the extension of the concept 'unicorn' is the empty set. Yet, the intended meaning of the concept is reasonably clear, which has much to do with the fact that people are generally able to form a 'mental image' of a unicorn (see fig. 1). In contrast, the extension of the concept 'square circle' is empty; however, while these two concepts are extensionally equal, they do not have the same intension.

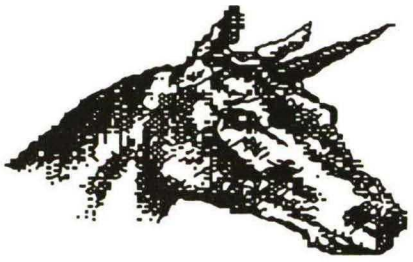


Figure 1. A unicorn

In this text we will use the language of set-theory and lattices when dealing with extensions of concepts: e.g., the extension of the concept 'unicorn' is ∅ or 0. Intensions of concepts or *concept descriptions* are described using a logic-based language that will not be defined formally. Sans serif typeface is reserved for expressions of this language: e.g., the intension of the concept 'square circle' could be shape(X) is square and shape(X) is circle. When dealing with concepts informally, quotes are used, as above in «the concept of 'unicorn'». In the course of the discussion, it will prove fruitful to distinguish a third level: the level of *concept representations*. Concept representations are implementations of concept intensions, and are indicated by the `typewriter font`. When we discuss our Prolog implementation of a simple concept learner, we will also use this font.

1.3 Organisation of the paper

The paper is organised as follows. In section 2, we present and discuss various ways of representing concepts. Because there are some difficulties associated with learning the general class of concepts, we limit attention to the more restricted class of simple concepts, and we present a way of representing simple concepts by Prolog terms called simple terms. It is shown that the set of simple terms is a Boolean lattice, and a Prolog-implementation of the lattice operations is given. In section 3, we discuss the semantics of simple terms, first intuitively using set-theory, then formally using abstract algebra. It appears that there is a mismatch in algebraic structure between the syntactic domain (simple terms) and the semantic domain (simple concepts), and we show how to map the former to the latter. In section 4, we address the problem of inductive learning of simple concepts using the representation of section 2. We finally derive several complete Prolog implementations of a concept learner, for (a) learning from positive examples only, (b) additionally checking consistency with negative examples, and (c) generating hypotheses also based on the negative examples. In section 5 we illustrate the operation of these programs by means of an example.

Most results of the more formal parts of this paper are stated as Theorems. Their proofs are given in an Appendix.

2. Representing simple concepts

2.1 Introduction

Informally, a concept can be defined as any set of objects that can be described by means of properties and their values. For instance, the concept of 'green circles' can be described by an expression like colour(X) is green and shape(X) is circle. A particular instance of this concept can be described by turning the variable X into a constant, say a, and perhaps adding some peculiarities of that particular object: colour(a) is green and shape(a) is circle and weight(a) is heavy. The problem of concept learning from examples is to find a concept description given positive and negative examples (i.e., descriptions of instances and non-instances of the concept). E.g., the above concept description can be induced from the description of instance a and the example colour(b) is green and shape(b) is circle and weight(b) is light. From this we see that both instance descriptions and concept descriptions may be cast into a vector representation such as

```
[colour(green),shape(circle),weight(heavy)]
```

for the first example, and

```
[colour(green),shape(circle),weight(_)]
```

for the induced concept, with the underscore _ denoting a DON'T CARE-value for the property weight.

This vector representation of concepts and instances indicates, that the induced concept vector may be considered *more general than* the given example vectors (in a sense soon to be made precise). In case negative examples are included, the concept vector should not be more general than any negative example vector. The above illustration also shows, that there may exist algebraic operators that take representations of positive examples and return a concept representation that is more general (similarly for negative examples). In this paper, we adopt this algebraic approach towards representing and learning concepts. Following this approach, a concept learner can be developed in three steps:

(*i*) Choose a syntactic representation for concepts and instances and determine the algebraic properties of this representation;

(*ii*) Describe the semantics of this representation algebraically;

(*iii*) Implement the concept learner by means of the syntactic operations defined in step (*i*); in case there is a mismatch in algebraic structure between the syntactic domain and the semantic domain, adjust these operations such that they correspond to the algebraic operations of the semantic domain.

In the case of the vector representation above, this three-step approach works out as follows. It has already been shown, that two vectors are generalised by matching them, introducing DON'T CAREs for conflicting property values. Likewise, two vectors can be specialised by a similar matching operation, whereby a property value overrules a DON'T CARE, and matching fails if two property values conflict. E.g., specialising

```
[colour(red),shape(_),weight(heavy)]
```

and

```
[colour(_),shape(circle),weight(heavy)]
```

yields

```
[colour(red),shape(circle),weight(heavy)];
```

specialisation of

```
[colour(red),shape(_),weight(heavy)]
```

and

```
[colour(blue),shape(circle),weight(heavy)]
```

fails. If we consider failure to specialise as the zero vector, then the set of vectors forms a complete lattice[1]. In fact, a vector can be interpreted as a first-order term (with DON'T CARE denoting an anonymous variable, as in Prolog), with unification playing the role of specialisation, and its dual operation anti-unification acting as generalisation.

The second step consists of specifying a semantics for vector representations, and analysing the algebraic properties of this semantic domain. This step will not be worked out now[2], but if we assume that properties are independent (i.e., any property can take on any value, regardless of the values of other properties), then the semantic domain is isomorphic to the syntactic domain. This means that we can use the syntactic operations of unification and anti-unification directly to implement a concept learner using vector representations (the third step).

2.2 Representing simple concepts

Vector representations as introduced in the preceding paragraph are not very expressive. They can only represent *conjunctive concepts*, i.e. concepts described by a logical conjunction (and) of property values. It would be nice to have a representation for a wider class of concepts, allowing logical disjunction (or) and negation (not). Such a concept may be described by an expression like (colour(X) is green and shape(X) is square) or weight(X) is not heavy. This has as effect that the most obvious representations for such Boolean concepts are not isomorphic to the corresponding semantic domain. While this is not really a problem (it only makes the task of building a concept learner more elaborate), there is a problem involved with **learning** such rich concepts. If there is no restriction on the logical form of concept descriptions, then the logical disjunction of the positive examples is **always** a valid concept description. Such a description is nothing more than a complete summary of the positive examples given, while in learning from examples we expect some **generalisation** to take place. This is the well-known *problem of disjunction*: allowing unrestricted disjunction in concept descriptions degenerates the learning capabilities of a concept learner.

This problem can be avoided by banning disjunction and complementation altogether from concept descriptions, as illustrated above. However, a less radical solution is obtained by allowing disjunction and negation only within one property. E.g., under this regime colour(X) is green or red and shape(X) is not square is allowed, while colour(X) is green or shape(X) is circle is not allowed. This form of disjunction is sometimes referred to as *internal disjunction* [Michalski 1980]; we prefer to use the term *simple concept* (description), in accordance with [Banerji 1969], who first suggested this solution to the disjunction problem.

We can adapt the vector representation to account for simple concepts as follows. In a concept representation, every property is followed by a list of values it may assume for that particular concept. We suppose that every property allows a fixed, finite number of values, so this list will be finite for every concept. Furthermore, we assume a fixed ordering on the set of values of each property (just as we assumed a fixed ordering of properties for vector representations). Finally, every value a property does not assume is represented by an underscore (or anonymous variable). So the following representation is valid:

---

[1] A *lattice* is a partially ordered set, where each two elements have unique least upper and greatest lower bounds; a *complete* lattice has one greatest and one smallest element.

[2] This can easily be done by assuming the usual set-theoretical semantics, with properties acting as **partitions** on the universe of discourse; see section 3.

```
[colour(_,blue,red,_),shape(_,square,_),weight(light,heavy)]
```

supposing the possible colours are green, blue, red and black, the possible shapes are triangle, square and circle, and the possible weights are light and heavy (also indicating their order). This list represents the concept description colour(X) is blue or red and shape(X) is square. Note, that the representation of a DON'T CARE value for a particular property is the enumeration of every possible value for that property (as in `weight(light,heavy)`). Furthermore, notice that this representation still facilitates the use of unification and anti-unification: e.g., the unification of the above term and

```
[colour(_,blue,_,black),shape(_,_,circle),weight(light,_)]
```

is the term

```
[colour(_,blue,red,black),shape(_,square,circle),weight(light,heavy)]
```

Notice carefully, that this term represents a more **general** concept than both original terms! That is, unification now corresponds to generalisation of concepts, and anti-unification now corresponds to specialisation of concepts.

We will use the above representation of simple concepts to implement a simple concept learner in Prolog. Instead of the recursive list functor (as suggested by the brackets `[]` above) we will use a fixed functor `st`, because the number of arguments is fixed (equal to the number of properties involved). The resulting Prolog terms are called *simple terms*, and the set of simple terms is denoted *ST*. An example of a simple term is

```
st(colour(_,blue,_,black),shape(_,_,circle),weight(light,_))
```

The arguments of the functor `st`, such as `colour(_,blue,_,black)` are called *simple disjunctive terms*.

2.3 The lattice of simple terms

It has been noted [Reynolds 1970, Plotkin 1970, 1971, Shapiro 1981], that the set of first-order terms in clausal logic forms a lattice, with unification and its dual anti-unification as join and meet (or conversely). In the general case, this lattice is infinite and non-modular. However, it is easily demonstrated that the set of simple terms *ST* constitutes a much more restricted lattice.

THEOREM 1. *The lattice ST of simple terms is a finite Boolean lattice.*

It is evident that a concept learner using simple terms should implement the operations of this Boolean lattice. It is less evident that this implementation should be based on Prolog terms. The two main reasons for making this choice have been:

(*i*) Simple terms in Prolog remain close to concept intensions, as opposed to, say abstract Boolean algebra;

(*ii*) Implementation in Prolog gives us the generalisation operation for free (unification).

Of course, we could have written unification and anti-unification algorithms in, say, Lisp [Luger & Stubblefield 1989]. A third reason for using Prolog was, to see how easy it could be done, and what the result would look like. Obviously, we are not using Prolog in the standard sense of declarative logic programming. In using Prolog's built-in unification mechanism, we have to take Prolog's operational semantics into account. We were curious whether this would affect the whole program, or just some core parts of it.

For constructing the most specific generalisation of two simple terms, we may simply unify them. Thinking operationally, we need a function returning the unification of two simple terms. However,

Prolog's built-in unification mechanism is a two-place predicate `unify`[3], resulting in a loss of the two original arguments. Therefore, these arguments are copied first by means of the predicate `copy_term`[4].

msg(ST1,ST2,ST3) ← *ST3* is the most specific generalisation of *ST1* and *ST2*

```
msg(ST1,ST2,ST3) :-
      copy_term(ST1,ST3),
      copy_term(ST2,ST4),
      unify(ST3,ST4).
```

For constructing the most general specialisation of two simple terms, we need the operation of anti-unification. Our variant of anti-unification, which we will call 'antification', will be implemented by means of the Boolean operation of complementation[5], which we will call 'complification'. It will prove fruitful to have complification at our disposal.

mgs(ST1,ST2,ST3) ← *ST3* is the most general specialisation of *ST1* and *ST2*

```
mgs(ST1,ST2,ST3) :-
      antify(ST1,ST2,ST3).
```

antify(ST1,ST2,ST3) ← *ST3* is the antification of *ST1* and *ST2*

```
antify(ST1,ST2,ST3) :-
      complify(ST1,ST10),
      complify(ST2,ST20),
      unify(ST10,ST20),
      complify(ST20,ST3).
```

com(ST1,ST2) ← *ST2* is the complement of *ST1*

```
com(ST1,ST2) :-
      complify(ST1,ST2).
```

complify(ST1,ST2) ← *ST2* is the complification of *ST1*

```
complify(ST1,ST2) :-
      ust(UST),
      complify(UST,ST1,ST2).
```

complify(ST1,ST2,ST3) ← *ST3* is the complification of *ST2* relative to *ST1*

```
complify(U,X,Y) :-
      var(X),unify(Y,U).
complify(U,X,Y) :-
      nonvar(X),atomic(X),var(Y).
complify(U,X,Y) :-
      nonvar(X),compound(X),term_cfy(U,X,Y).
```

[3]In fact, unification without occurs check (predicate =) suffices in our case.
[4][Sterling & Shapiro 1986], p. 180, predicate `copy`.
[5]Therefore, antification is not equivalent to anti-unification in the general non-Boolean lattice of first-order terms.

```
term_cfy(U,X,Y) :-
      functor(U,F,N),functor(X,F,N),
      args_cfy(N,0,U,X,Ys),
      Y=..[F|Ys].

args_cfy(N,M,U,X,[ArgY|Ys]) :-
      M<N,M1 is M+1,arg_cfy(M1,U,X,ArgY),
      args_cfy(N,M1,U,X,Ys).
args_cfy(N,N,U,X,[]).

arg_cfy(N,U,X,ArgY) :-
      arg(N,U,ArgU),arg(N,X,ArgX),
      complify(ArgU,ArgX,ArgY).
```

Complification is taken relative to the universal simple term, given by the predicate `ust`. This predicate is dependent on the specific learning task at hand, and should be user-defined. For the examples we gave in paragraph 2.2, we have

```
ust(st(colour(green,blue,red,black),
       shape(triangle,square,circle),
       weight(light,heavy))).
```

The three-place predicate `complify` is, with a few minor modifications, analogous to the explicit unification algorithm given in [Sterling & Shapiro 1986], p. 150.

The declarative meaning of the predicates `msg`, `mgs` and `com` (and, for that matter, `antify` and `complify`) corresponds to their operational meaning only for a specific use of arguments. The reason is, that these predicates in fact play the role of functions. More specifically, each of these predicates (as they are now) should only be called with all but the last argument instantiated. These restrictions may cause trouble, if we want to use join, meet and complement to implement partial ordering:

```
ST1 ≤ ST2 :- mgs(ST1,ST2,ST1).
```

or express disjointness:

```
disjoint(ST1,ST2) :-
      ust(UST),complify(UST,NST),mgs(ST1,ST2,NST).
```

This can be remedied by testing whether the last argument is instantiated, as follows.

```
msg(ST1,ST2,ST3) :-
      copy_term(ST1,ST4),
      copy_term(ST2,ST5),
      unify(ST4,ST5),
      check(ST3,ST4).

mgs(ST1,ST2,ST3) :-
      antify(ST1,ST2,ST4),
      check(ST3,ST4).

com(ST1,ST2) :-
      complify(ST1,ST3),
      check(ST2,ST3).
```

check(ST1,ST2) ← *ST1* and *ST2* are equivalent simple terms (*ST2* should be instantiated)

```
check(ST1,ST2) :-
      var(ST1),unify(ST1,ST2).
check(ST1,ST2) :-
      nonvar(ST1),equiv(ST1,ST2).
```

equiv(ST1,ST2) ← *ST1* and *ST2* are equivalent simple terms (*ST1* and *ST2* should be instantiated)

```
equiv(X,Y) :-
      var(X),var(Y).
equiv(X,Y) :-
      nonvar(X),atom(X),
      nonvar(Y),atom(Y),
      unify(X,Y).
equiv(X,Y) :-
      nonvar(X),compound(X),
      nonvar(Y),compound(Y),
      equiv_terms(X,Y).

equiv_terms(X,Y) :-
      functor(X,F,N),functor(Y,F,N),
      equiv_args(N,X,Y).

equiv_args(N,X,Y) :-
      N>0,equiv_arg(N,X,Y),
      N1 is N-1,equiv_args(N1,X,Y).
equiv_args(0,X,Y).

equiv_arg(N,X,Y) :-
      arg(N,X,ArgX),arg(N,Y,ArgY),
      equiv(ArgX,ArgY).
```

If `check` finds that the last argument of `msg`, `mgs` and `com` is not instantiated, it is simply unified with the desired result. If it is instantiated, it is tested for equivalence with the desired result by means of the predicate `equiv`, which is again defined in rather general terms.

3. The semantics of simple terms

3.1 Introduction

In the preceding section, we discussed the issue of how to represent simple concepts. Such a representation is a purely syntactical matter, and operations on such representations, such as matching and unification, are purely syntactical operations. Accordingly, representations that are syntactically different are really different objects. But do syntactically different representations always represent different things? And, do syntactic operations like unification and anti-unification always correspond to semantic operations like generalisation and specialisation?

To answer questions like these, we must turn to the *semantics* of simple terms, and we will do so in this section. The semantics will first be described informally, by means of notions from set-theory. After that, it will be developed more formally, in abstract algebraic terms. We will discover a mismatch in algebraic structure between the syntactic domain of simple terms and this semantic domain. Subsequently, in section 4 we will describe ways to accomodate for this mismatch in the concept learner that is being built.

While syntax has to do with intensions of concepts, semantics concerns extensions. In order to avoid introducing yet another set of technical terms, we will henceforth use the term *concept* as a technical term, denoting an extension (i.e., a set of objects). Likewise, we will speak about *properties*, *values*, and *simple concepts* as denoting formal set-theoretical or algebraic objects.

3.2 An informal semantics

In this paragraph, we follow [Banerji 1969] to a large extent. When describing concepts extensionally, we will only consider objects from a predefined *universe* U. Thus, any concept is a subset of U. Moreover, we assume a fixed and finite collection $\mathcal{P}=\{P_1,...,P_n\}$ of properties P_i. The basic step towards a semantics for simple terms is to consider each *property* as a **partition** on the universe. For instance, in a universe where each element has exactly one colour, while there are a finite number of colours, colour in fact forms a partition on the universe, inducing an equivalence relation, i.e. elements with the same colour are indiscernible from each other as far as their colour is concerned. The *values* of the property correspond to the blocks of the partition.

Adding more properties obviously increases the 'degree of discernibility': if we add the property shape to the property colour, we can distinguish between red circles and red squares. In terms of partitions, this is equivalent to saying that two partitions can be combined to form a finer partition. All available properties taken together form a finest partition, i.e. even though the universe may contain an infinite number of objects, $\mathcal{P}$ represents a 'grid' of finite[6] granularity, and thus some elements of U may remain indistinguishable from some others. The set $\mathcal{P}$ is referred to as an *environment* (U is understood implicitly).

Concepts are formed by combining values of properties in several ways: we can speak of objects that are either green or red, of blue circles (i.e. objects that are both blue and circles), and of objects that are not heavy. Following this observation, a *concept* is defined to be any set that can be construed from values of properties using the set-theoretical operators union, intersection and complement. We can further

[6]It is also assumed, that each property is a **finite** partition (containing a finite number of blocks).

distinguish between arbitrary, conjunctive, and simple concepts. A *conjunctive concept* can only be construed from property values using intersection, while a *simple concept* is equal to the intersection of some simple disjunctive concepts, where each *simple disjunctive concept* is the union of some values of one property. Any set of concepts can be partially ordered by set-inclusion. This ordering corresponds to the notion of **generality**: a concept is at least as general as another concept if it contains at least the same objects.

It is well-known that the powerset (the set of subsets) of any set forms a Boolean algebra. The set of arbitrary concepts is a subalgebra of the powerset of U, because it is closed under union and intersection; hence the set of arbitrary concepts is also a Boolean algebra. The set of simple concepts (henceforth denoted SC) is not a Boolean algebra, as will be shown in the next paragraph. We will see that SC 'almost looks like' a Boolean algebra in the upward direction, while irregularities occur in the downward direction.

3.3 The lattice of simple concepts

Although the initial definition of concepts was stated in terms of sets, we proceed our exposition by using more abstract algebraic symbols. In particular, 0 is used for the empty concept, 1 for the universe concept; a property constitutes the valueset of a multivalued Boolean variable. The usual Boolean laws for the algebra of subsets of a universe set are assumed, for example $X \leq Y$ iff $X \wedge Y = X$, $X \wedge X' = 0$, $X \vee X' = 1$, etc. In addition, the following abbreviations are used: $\wedge_i X_i$ for $X_{i1} \wedge X_{i2} \wedge \ldots \wedge X_{in}$, where the index set $\{i1,i2,\ldots,in\}$ is understood implicitly; likewise, $\{X_i\}$ stands for $\{X_{i1},X_{i2},\ldots,X_{in}\}$.

Consider the task of finding a description of a given simple concept S. To accomplish this, we have to construct a simple disjunctive concept S_i for each property P_i, such that the conjunction of these disjunctions yields the original simple concept: $\wedge_i S_i = S$. The question now is: is there more than one way to do this? Unfortunately, the answer to this question is affirmative. Consider for instance the empty concept 0: as long as at least one simple disjunctive concept $S_k = 0$, their conjunction is equal to 0. Alternatively, suppose that the properties colour and shape are interrelated in the following sense: every blue object in the universe is a circle. Put differently: if we know an object is blue, then it is a circle. In this case, we may say that the property shape is *dependent on* the property colour. Now suppose that S is a simple concept containing only blue objects, then we may include **any** shape besides circle in the simple disjunctive concept corresponding to the property shape, and still obtain a valid description for S (augmenting S with an empty set[7] yields S).

We must conclude that, in the general case, a given simple concept has several descriptions. However, each simple concept has a unique **minimal** description, such that every simple disjunctive concept in that description is minimal (no property value can be removed from it). The set $\{S_i\}$ such that for every i, S_i is a simple disjunctive concept for property P_i, and $\wedge_i S_i = S$, is called a *decomposition* of S. We thus have the following theorem.

THEOREM 2. *Every simple concept has a unique minimal decomposition.*

Let $\{S_i\}$ be a minimal decomposition for S, then each simple disjunctive concept S_i is called the *projection of S on* P_i. The notation S_i will be reserved for the projection of S on P_i. E.g., $0_i = 0$ for any i (the projection of the empty concept 0 on any property is 0).

Every property determines a set of simple disjunctive concepts. Each of these sets in fact forms a Boolean algebra. Decompositions can be seen as the elements of the direct product of the simple disjunctive concept-algebras. The notion of a **minimal** decomposition is necessary, because

[7] Such as the set of blue non-circles.

decompositions do not uniquely determine simple concepts. Just as the ordering in factor algebras is preserved in the direct product, the ordering between projections is preserved in SC. This is expressed in Theorem 3, which will prove instrumental in revealing the structure of SC. The proof makes repeatedly use of the fact that, while $S=\wedge_i S_i$, for all i $S\leq S_i$. Moreover, if T_k is a simple disjunctive concept for property P_k and S is a simple concept, $T_k \wedge S=0$ implies $T_k \wedge S_k=0$ (the values in T_k are *not relevant for* S, and are not contained in the projection of S on P_k).

THEOREM 3. *For any two simple concepts S and T, $S\leq T$ iff for all i, $S_i \leq T_i$;*

The significance of Theorem 3 is, that (just as in a direct product) join and meet of projections can be used to construe join and meet of simple concepts.

THEOREM 4. *$\langle SC,\leq\rangle$ is a lattice. Let S and T be simple concepts, their meet is given by $V=\wedge_i(S_i \wedge T_i)$ and their join is given by $W=\wedge_i(S_i \vee T_i)$. For all i, $W_i=(S_i \vee T_i)$, but V_i not necessarily equals $(S_i \wedge T_i)$.*

The join operation for simple concepts will be called *simple disjunction*, and the meet operation will be called *simple conjunction*. When no confusion can possibly arise, the lattice $\langle SC,\leq\rangle$ will also be denoted by its carrier set SC. Theorem 4 states, that the operation of simple disjunction has a Boolean nature (it behaves just like the join operation in the direct product of the simple disjunctive concept algebras), while Boolean laws are violated in the downward direction (from general to specific) of simple conjunction. E.g., if $S \wedge T=0$, then for all i $V_i=0$, but there may be a k for which $S_k \wedge T_k \neq 0$, which means that the minimal decomposition of V can not directly be derived from the minimal decompositions of S and T.

It has to be noted that the simple conjunction of disjoint simple concepts is not the only possible case in which a non-minimal decomposition can arise. We have seen above that properties may be interrelated, such that some non-empty simple concepts have more than one decomposition. If we do not want to admit such cases, we may insist that properties are mutual independent (one property can take on any of its values, regardless of the values for the other properties), resulting in a *full* environment. There are several ways to define this notion: here we use the definition, that in a full environment every non-empty simple concept has exactly one decomposition[8]. If an environment is not full, additional information has to be provided, describing the dependencies between properties; this we call *background knowledge*[9]. The concept learner that will be implemented only works for full environments; however, most of the results in this section are also valid for non-full environments.

We proceed by briefly stating the main properties of the lattice SC. We denote the simple conjunction of two simple concepts S and T by $S \otimes T$, and their simple disjunction by $S \oplus T$.

[8] An equivalent definition states, that the number of blocks in the finest partition that can be built out of properties, is equal to the product of the numbers of values of each property.
[9] See [Flach & Veelenturf 1989] for a discussion of this kind of background knowledge.

THEOREM 5. *Let $\mathcal{P}$ be an environment containing at least two properties, each with at least two values.*

(a) *SC is atomic.*

(b) *SC is complemented, although not uniquely.*

(c) *SC is non-modular.*

(d) *SC is not semi-modular.*

(e) *SC is not distributive.*

(f) *SC is not relatively complemented.*

There seems to be an obvious way to define a special unique simple complement T of a simple concept S, such that for all i, (*i*) $S_i \wedge T_i=0$, and (*ii*) $S_i \vee T_i=1$, i.e. $T=\wedge_i(S_i)'$. Things are, however, not as simple as it may seem, because if some $S_k=1$, then $(S_k)'=0$, hence $T=0$, contradicting $S \oplus T=1$ if $S<1$. Hence, the simple complement of any simple concept S is defined as $\Theta S=\wedge_i \neg(S_i)$, where $\neg X=X'$ if $X<1$, and $\neg X=1$ otherwise. Again, we can only be sure that $\neg(S_i)$ denotes a projection of ΘS if the environment is full.

In the Appendix, some additional properties of the lattice *SC* are given. Now that we have analysed the semantic domain algebraically, we can define the relation between the syntactic domain of simple terms and the semantic domain of simple concepts.

3.4 Mapping simple terms to simple concepts

The mapping m (for meaning) from simple terms to simple concepts should satisfy some requirements. First of all, m should be such that every simple term represents at most one simple concept, and every simple concept is represented by at least one simple term (the mapping m is a surjective function). Additionally, it may be required that

(*i*) there is a quasi-ordering on the set of simple terms, corresponding to the partial generality ordering $\leq$ on *SC*;

(*ii*) every simple term represents exactly one simple concept;

(*iii*) every simple concept is represented by exactly one simple term.

If (*i*) is valid, m is called an *order homomorphism*; if additionally (*ii*) is valid, m is an *order quasi-isomorphism*; if additionally (*iii*) is valid (hence, m is a bijection), m is an *order isomorphism* [Laird 1988]. Requirement (*i*) is very important, for it allows a learner to utilise the semantic generality ordering when manipulating simple terms, by means of the syntactic quasi-ordering, as will be explained below.

The definition of m is not difficult to state. Each simple disjunctive term in a simple term corresponds to the simple disjunctive concept of the property values occurring as constants in the simple disjunctive term. Consequently, a simple term can be interpreted as the decomposition of a simple concept. We now have the following Theorem.

THEOREM 6. *The meaning function m is an order quasi-isomorphism. In particular, if ST3 is the unification of ST1 and ST2, then $m(ST1) \oplus m(ST2)=m(ST3)$; and if ST4 is the antification of ST1 and ST2, then $m(ST1) \otimes m(ST2)=m(ST4)$.*

The fact that m is not an order isomorphism means, that one simple concept may be represented by several simple terms. Conversely, several simple terms may have the same meaning (image under m). Thus, *ST* can be partitioned into equivalence classes, each element of an equivalence class having the same meaning; moreover, these equivalence classes can be partially ordered, according to the ordering $\leq$ on *SC*. Each equivalence class contains exactly one member corresponding to a minimal decomposition: this simple term will be called the *correct representation* of the simple concept that is its meaning.

COROLLARY 7. *If ST1 is a correct representation for m(ST1), and ST2 is a correct representation for m(ST2), then the unification of ST1 and ST2 is a correct representation of m(ST1)⊕m(ST2), but the antification of ST1 and ST2 is possibly not a correct representation of m(ST1)⊗m(ST2).*

In the following section, we will show how to change the Prolog predicates in order to account for this problem.

4. Learning simple concepts

4.1 Introduction

In this section, we restrict ourselves to **full** environments. In this way, we can be sure that every representation for a non-empty simple concept is a correct representation. Consequently, we will only have to bother about simple terms whose meaning is 0. Anyhow, this problem will only be encountered when proceeding from general concepts to specific concepts. It will not occur if we are learning from positive examples only, because in this case only the operation of generalisation is needed. Indeed, within the framework outlined in the previous sections, a very simple implementation of learning from positive examples is possible.

Let us start by outlining the simple concept learning problem in somewhat more detail. A simple concept learner is given positive and negative examples of the simple concept to be learned; its task is to determine the target simple concept. Now, any simple concept which is to be considered a candidate target simple concept, should satisfy the consistency conditions. A simple concept is *consistent* with the examples if, for every positive example PE, $S \geq PE$, and for every negative example NE, S is disjoint from NE or $S \otimes NE = 0$. Suppose we have a set of examples SE and a simple concept S consistent with these examples. If we encounter a new example E such that S is not consistent with $SE \cup \{E\}$, then we must change our current hypothesis S. If E is a positive example, it follows that not $S \geq E$, and we must find a T such that $T \geq E$ and T is consistent with SE. Choosing $T \geq S$ will guarantee that T is consistent with every positive example in SE; choosing the smallest T possible will guarantee that if there exist simple concepts consistent with $SE \cup \{E\}$, T will be one of them. But the smallest T such that $T \geq S$ and $T \geq E$ is given by $T = S \oplus E$: thus, the operation of simple disjunction of the lattice SC can be used to generalise a hypothesis found to be not consistent with a new positive example.

Likewise, if E is a negative example, it follows that $S \otimes E \neq 0$ and we must find a V such that $V \otimes E = 0$ and V is consistent with SE. Choosing $V \leq S$ will guarantee that V is consistent with every negative example in SE; choosing the largest V possible will guarantee that if there exist simple concepts consistent with $SE \cup \{E\}$, V will be one of them. The largest V such that $V \leq S$ and $V \otimes E = 0$ is given by $V = S \otimes W$, where W is some maximal complement of E. Thus, the operation of simple conjunction of the lattice SC, in cooperation with any operation of simple maximal complementation, can be used to specialise a hypothesis found to be not consistent with a new negative example.

As a result of the foregoing observations, there exists a unique simple concept S, such that every consistent simple concept $T \geq S$: S is the simple disjunction of all positive examples. In general, there does not exist a unique simple concept G such that every consistent simple concept $T \leq G$. Such a G might be construed by taking the simple conjunction of one maximal complement of each negative example[10]. Because complements are not unique, there will be several such G's. As in the general case of Version Spaces [Mitchell 1982], S and the G's are bounds on the set of consistent hypotheses, and characterise this set completely, together with the ordering of the hypothesis space.

[10] Additionally, we have to check that $G \geq S$ (which is not necessarily the case).

4.2 Learning from positive examples only

Suppose we want to learn a simple concept from positive examples only; that is, there are no negative examples available to prevent overgeneralisation. It can be seen, that the consistency condition formulated above is not enough: the universal simple concept 1 will be consistent with any set of positive examples. Therefore, we should also consider measures of convergence (this can be formalised by the model of identification in the limit [Gold 1967]). In such a model a cautious learning strategy is required, generalising the current hypothesis only when forced to by a new example. That is, at every moment the hypothesis is the most specific one, i.e. *S*.

Within our current framework, the following very simple implementation of a simple concept learner using positive examples only is obtained. We assume that representations for positive examples are given by means of the predicate `pos_ex`.

learn(H) ← — *H* is a solution to the learning problem

```
learn(H) :-
      mshyp(H).
```

mshyp(H) ← — *H* is the most specific hypothesis

```
mshyp(H) :-
      setof(PE,pos_ex(PE),L),
      msg_list(L,H).
```

msg_list(L,G) ← — *G* is the most specific generalisation of the simple terms in the list *L*

```
msg_list([E|Es],G) :-
      msg_list(Es,G2),
      msg(E,G2,G).
msg_list([],NST) :-
      nst(NST).
```

nst(NST) ← — *NST* is the null simple term

```
nst(NST) :-
      ust(UST),com(UST,NST).
```

To repeat: this simple program works, because the lattice *SC* 'resembles' the Boolean algebra *ST* in the 'upward' direction of generalisation.

4.3 Learning from positive and negative examples

Things get more delicate when we take negative examples into consideration. However, note that it is still safe to guess the most specific hypothesis: the cautious strategy is still guaranteed to identify the target simple concept in the limit. We only have to add a test, that this hypothesis indeed satisfies the negative examples (if not, the learning problem is unsolvable). This method may seem a bit crude, because the information hidden in the negative examples is completely ignored (for instance, they might be 'near misses'). We will shortly consider methods to overcome this objection.

The top-level predicate `learn` is redefined as

```
learn(H) :- mshyp(H),testneg(H).
```

The predicate `testneg(H)` is true if `H` is consistent with every representation of a negative example; that is, for each negative example representation `NE`, $m(H) \otimes m(NE)=0$. Consider the following definition of `testneg`:

testneg(H) ← *H* is consistent with the negative examples

```
testneg(H) :-
      nst(NST),
      forall(neg_ex(NE),mgs(H,NE,NST)).
```

The problem here is, that the current implementation of `mgs` is not guaranteed to result in a correct representation, when the result represents 0, and that is exactly what is tested here. Therefore, the predicate `mgs` should be changed as follows.

```
mgs(ST1,ST2,ST3) :-
      antify(ST1,ST2,ST4),
      cr_mgs(ST4,ST5),
      check(ST3,ST5).
```

cr_mgs(ST1,ST2) ← *ST1* is a correct representation, unless it represents 0; *ST2* is a correct representation for the same simple concept

```
cr_mgs(ST1,ST2) :-
      is_null(ST1),nst(ST2).
cr_mgs(ST,ST) :-
      not is_null(ST).
```

is_null(ST) ← *ST* represents 0

```
is_null(ST) :-
      functor(ST,st,N),is_null_sdts(ST,N).

is_null_sdts(ST,N) :-
      is_null_sdt(ST,N).
is_null_sdts(ST,N) :-
      not is_null_sdt(ST,N),N>0,N1 is N-1,
      is_null_sdts(ST,N1).

is_null_sdt(ST,N) :-
      arg(N,ST,SDT),is_null_vals(SDT,N).

is_null_vals(SDT,0).
is_null_vals(SDT,N) :-
      N>0,arg(N,SDT,Val),var(Val),N1 is N-1,
      is_null_vals(SDT,N1).
```

We now have a simple concept learner for positive as well as negative examples, that finds a solution based on the positive examples only; negative examples are used only to check consistency of the solution.

4.4 Taking negative examples serious

Next, we consider an approach that is expected to be more faithful to the information contained in negative examples. The idea is, to take the simple complement of a negative example as a candidate hypothesis. The more the negative example differs from the target concept, the more its complement will agree with it; the more specific the negative example, the more general its complement. Note, that this approach interprets negative examples as 'far misses' rather than as 'near misses' [Winston 1975]. Because it is not guaranteed, that such a hypothesis is indeed consistent with the other examples, it is joined with the positive examples; therefore, we expect to get a more general hypothesis this way. As before, consistency with the remaining negative examples also has to be tested.

The following clause for `learn` is added:

```
learn(H) :- mghyp(H),testneg(H).
```

mghyp(H) ← *H* is a more general hypothesis

```
mghyp(H) :-
      neg_ex(NE),com(NE,Hneg),
      mshyp(Hpos),msg(Hpos,Hneg,H).
```

If we assume, that examples are given by most specific conjunctive concepts, listing exactly one value for each property, then the predicate `com` works correct. However, we still have to consider the case that some simple disjunctive term contains all values of a property, because in this case its simple complement has to contain the same simple disjunctive term (and not its complification). Therefore, we change the definition of `com` as follows.

```
com(ST1,ST2) :-
      complify(ST1,ST3),
      cr_com(ST1,ST3,ST4),
      check(ST2,ST4).
```

cr_com(ST1,ST2,ST3) ← *ST2* is a correct representation for $\Theta m(ST1)$, unless $m(ST2)=0$ while $m(ST1)\neq 1$; *ST3* is a correct representation for the same simple concept $\Theta m(ST1)$ (*ST1* and *ST2* should be instantiated)

```
cr_com(ST1,ST2,ST2) :-
      not is_null(ST2).
cr_com(ST1,ST2,ST2) :-
      is_null(ST2),ust(ST1).
cr_com(ST1,ST2,ST3) :-
      is_null(ST2),not ust(ST1),
      functor(ST1,st,N),functor(ST2,st,N),
      cr_com_sdts(N,0,ST1,ST2,SDT3s),
      ST3 =.. [st|SDT3s].

cr_com_sdts(N,M,ST1,ST2,[SDT1|SDT3s]) :-
      M<N,M1 is M+1,
      arg(M1,ST2,SDT2),is_null_sdt(SDT2),
      arg(M1,ST1,SDT1),
      cr_com_sdts(N,M1,ST1,ST2,SDT3s).
```

```
cr_com_sdts(N,M,ST1,ST2,[SDT2|SDT3s]) :-
      M<N,M1 is M+1,
      arg(M1ST2,SDT2),not is_null_sdt(SDT2),
      cr_com_sdts(N,M1,ST1,ST2,SDT3s).
cr_com_sdts(N,N,ST1,ST2,[]).
```

With this enhanced version of `com`, we can finally introduce a third, medium form of hypothesis: the simple complement of the simple disjunction of all negative examples. This hypothesis may be interpreted as a summary of all the negative examples. On the other hand, it is **not** a most general hypothesis, because the simple disjunction of all negative examples is at least as general as each negative example; consequently, the simple complement of this simple disjunction is at most as general as the simple complement of each negative example (the more general hypotheses introduced earlier). For guaranteeing concistency with the positive examples, this hypothesis has to be joined with the positive examples. Because this operation in turn might introduce an inconsistency with a negative example, it has to be tested again for consistency with the negative examples.

A third clause for `learn` is added:

```
learn(H) :- medhyp(H),testneg(H).
```

medhyp(H) ← *H* is a medium hypothesis

```
medhyp(H) :-
      setof(NE,neg_ex(NE),L),
      msg_list(L,G),com(G,Hneg),
      mshyp(Hpos),msg(Hpos,Hneg,H).
```

Calling `learn(H)` will now result in one most specific solution, as many more general solutions as there are negative examples, and one medium solution (some of these may not really exist).

5. A session with the simple concept learner

The operation of the simple concept learner just described is illustrated with an example. An environment {*colour,shape,size*} is assumed, consisting of the properties *colour*={*red,green,blue,black*}, *shape*={*circle,oval,rectangle,triangle*}, *size*={*big,small*}. This environment is described by means of the predicate `ust`, as follows.

```
ust(st(colour(red,green,blue,black),
       shape(circle,oval,rectangle,triangle),
       size(big,small)).
```

As positive examples, we take a big red circle and a big green rectangle; as negative examples, we take a small blue circle and a small black oval. The examples are specified by means of the predicates `pos_ex` and `neg_ex`.

```
pos_ex(st(colour(red,_,_,_),
          shape(circle,_,_,_),
          size(big,_)).

pos_ex(st(colour(_,green,_,_),
          shape(_,_,rectangle,_),
          size(big,_)).

neg_ex(st(colour(_,_,blue,_),
          shape(circle,_,_,_),
          size(_,small)).

neg_ex(st(colour(_,_,_,black),
          shape(_,oval,_,_),
          size(_,small)).
```

The goal `?-learn(H)` will result in the following answers:

```
H =  st(colour(red,green,_3177,_3178),
        shape(circle,_3182,rectangle,_3184),
        size(big,_3188));

H =  st(colour(red,green,_4124,black),
        shape(circle,oval,rectangle,triangle),
        size(big,_4135));

H =  st(colour(red,green,blue,_4125),
        shape(circle,_4129,rectangle,triangle),
        size(big,_4135));

H =  st(colour(red,green,_6321,_6322),
        shape(circle,_6326,rectangle,triangle),
        size(big,_6332));
```

```
No more solutions
```

Let us trace these results in somewhat more detail.

The first solution, 'big circles or rectangles that are green or red', is the most specific one, and is readily obtained from the positive examples alone, by means of the predicate `mshyp`. The second solution states that anything that is not blue and is big belongs to the target simple concept. It is obtained from the first negative example, of which the complementation yields

```
st(colour(red,green,_,black),
   shape(_,oval,rectangle,triangle),
   size(big,_))
```

Unifying this term with the join of the positive examples (the first solution) yields the second solution. The third solution is likewise obtained from the second negative example on backtracking (hence the recurrence of the variable `_4135`). Complementation of the second negative example yields

```
st(colour(red,green,blue,_),
   shape(circle,_,rectangle,triangle),
   size(big,_))
```

Notice, that this term subsumes the join of the positive examples; hence, unifying it with the latter does not add anything new. Finally, the fourth solution is obtained from all four examples, by means of the predicate `medhyp`. Complementation of the join of the negative examples yields

```
st(colour(red,green,_,_),
   shape(_,_,rectangle,triangle),
   size(big,_))
```

Unification of this term with the first solution yields the fourth solution, which is (in this case) the meet of the second and the third. On a future occasion, we will report on the different uses that can be made of negative examples, and the relations between these uses.

6. Concluding remarks

A simple concept learner has been developed, implemented in Prolog. We arrived at this perhaps somewhat unusual program by means of a thorough formal investigation of the algebraic structures involved. Nevertheless, we are confident that this approach has revealed some difficulties involved with inductive learning.

Our choice for Prolog has been motivated by the need for lattice operations, one of which was implemented by unification. The dual operation of anti-unification had to be programmed explicitly, a task for which Prolog is not especially suited (though it can easily be done). Another drawback of this approach is, that the declarative meaning of some predicates is readily destroyed by calling them with some variables instantiated. A large part of our efforts has been devoted to the repair of such phenomena. As a result, the non-declarative parts of the program are limited in number, and more or less hidden from the higher levels.

It would perhaps be more convenient, to implement a system with unification and anti-unification in a lower-level language, and to use this system to implement our concept learner. Of course, there are several other efficient implementations of lattice operations [Ait-Kaci *et al.* 1989]. Nevertheless, we think the implementation presented here has some elegance, perhaps illustrated best by the way it has been developed (by means of formal methods), and by the fact that a complete listing has been given in this paper.

Finally, we mention an interesting extension of the current work by means of Ψ-terms (Ait-Kaci & Nasr 1986]. This formalism permits the unification of terms with different functors, if one functor is declared to be more general than the other by means of a hierarchy. The relevance of this idea to the field of Machine Learning is even more obvious, if we recall that a predicate can be viewed as a functor on the meta-level.

Acknowledgements

The use of unification and anti-unification was suggested to the author by Rene Ahn.

References

[Ait-Kaci & Nasr 1986] H. AIT-KACI & R. NASR, 'LOGIN: A logic programming language with built-in inheritance', *Journal of Logic Programming* 1986:3, 185-215.

[Ait-Kaci *et al.* 1989] H. AIT-KACI, R. BOYER, P. LINCOLN & R. NASR, 'Efficient implementation of lattice operations', *ACM Transactions on Programming Languages and Systems* 11:1, 115-146.

[Banerji 1969] R.B. BANERJI, *Theory of problem solving: an approach to Artificial Intelligence*, Elsevier, New York.

[Flach & Veelenturf 1989] P.A. FLACH & L.P.J. VEELENTURF, *Concept learning from examples: theoretical foundations*, ITK Research Report 2, Institute for Language Technology and Artificial Intelligence, Tilburg University, Tilburg, the Netherlands.

[Gold 1967] E.M. GOLD, 'Language identification in the limit', *Information and Control* 10, 447-474.

[Laird 1988] P.D. LAIRD, *Learning from good and bad data*, Kluwer.

[Luger & Stubblefield 1989] G.F. LUGER & W.A. STUBBLEFIELD, *Artificial Intelligence and the Design of Expert Systems*, Benjamin/Cummings.

[Michalski 1980] R.S. MICHALSKI, 'Pattern recognition as rule-guided inductive inference', *IEEE Transactions on Scene Analysis and Machine Intelligence* 2:4, pp. 349-361.

[Mitchell 1982] T.M. MITCHELL, 'Generalization as search', *Artificial Intelligence* 18, 203-226.

[Plotkin 1970] G.D. PLOTKIN, 'A note on inductive generalisation', in *Machine Intelligence 5*, B. Meltzer & D. Michie (eds.), Edinburgh University Press, pp. 153-163.

[Plotkin 1971] G.D. PLOTKIN, 'A further note on inductive generalisation', in *Machine Intelligence 6*, B. Meltzer & D. Michie (eds.), Edinburgh University Press, pp. 101-124.

[Reynolds 1970] J.C. REYNOLDS, 'Transformational systems and the algebraic structure of atomic formulas', in *Machine Intelligence 5*, B. Meltzer & D. Michie (eds.), Edinburgh University Press, pp. 135-151.

[Shapiro 1981] E.Y. SHAPIRO, *Inductive inference of theories from facts*, Technical Report 192, Yale University.

[Sterling & Shapiro 1986] L. STERLING & E.Y. SHAPIRO, *The art of Prolog*, MIT Press, Cambridge, MA.

[Winston 1975] P.H. WINSTON, 'Learning structural descriptions from examples', in *The psychology of computer vision*, P.H. Winston (ed.), McGraw-Hill.

Appendix

A.1 Simple terms

THEOREM 1. *The lattice ST of simple terms is a finite Boolean lattice.*

Proof. Let v denote any property value, and consider the set $Val(\mathtt{v})=\{\mathtt{v},\_\}$. Treating v as a constant symbol and _ as a variable symbol, it is easily established that $Val(\mathtt{v})$ constitutes a Boolean lattice, with unification as join and anti-unification as meet (or *vice versa*). For any property P, assume some fixed ordering of its values, and let $Prop(\mathtt{P})$ be the Cartesian product of the sets $Val(\mathtt{v})$, in that order, for every value v of P. Writing P (...) for elements of $Prop(\mathtt{P})$ instead of ‹...›, we immediately recognise $Prop(\mathtt{P})$ as the set of simple disjunctive terms associated with property P. $Prop(\mathtt{P})$ is again a Boolean lattice, equal to the direct product of the Boolean lattices $Val(\mathtt{v})$, with again unification as join and anti-unification as meet, applied to an element of $Prop(\mathtt{P})$ by applying it to each coordinate separately. Finally, assuming some fixed ordering of properties, we can construe the direct product of the Boolean lattices $Prop(\mathtt{P})$, again yielding a Boolean lattice. This lattice is finite, provided both the number of properties and the number of values is finite. ■

A.2 Simple concepts

THEOREM 2 . *Every simple concept has a unique minimal decomposition.*

Proof. Every simple concept has at least one decomposition. Any non-minimal decomposition can be turned into a minimal one by repeatedly removing values from simple disjunctive concepts until any further removal would imply that it is no longer a decomposition for the intended simple concept. Suppose that there are two minimal decompositions $\{S_i\}$ and $\{T_i\}$ for a simple concept S, i.e. $S=\wedge_i S_i=\wedge_i T_i$; it follows that $S=\wedge_i(S_i\wedge T_i)$, and hence $\{S_i\wedge T_i\}$ is also a decomposition of S. From the minimality of $\{S_i\}$ and $\{T_i\}$ it follows that $S_i=T_i$ for all i, hence $\{S_i\}$ and $\{T_i\}$ are identical. ■

THEOREM 3. *For any two simple concepts S and T, $S\leq T$ iff for all i, $S_i\leq T_i$.*

Proof. The if part is obvious. For the only if part, suppose not $S_k\leq T_k$ for some k, implying $(S_k-T_k)\neq 0$. But it is always the case that $(S_k-T_k)\wedge T_k=0$; because $T\leq T_k$, $S\leq T$ implies that $(S_k-T_k)\wedge S=0$. But then (S_k-T_k) can be removed from S_k, contradicting the fact that S_k is the projection of S on P_k. ■

THEOREM 4. *‹SC,≤› is a lattice. Let S and T be simple concepts, their meet is given by $V=\wedge_i(S_i\wedge T_i)$ and their join is given by $W=\wedge_i(S_i\vee T_i)$. For all i, $W_i=(S_i\vee T_i)$, but V_i not necessarily equals $(S_i\wedge T_i)$.*

Proof. First, it is noted that V and W are conjunctions of simple disjunctive concepts, proving that they are simple concepts[11].
(meet) $V=\wedge_i(S_i\wedge T_i)=(\wedge_i S_i)\wedge(\wedge_i T_i)=S\wedge T$, which is the largest concept contained by both S and T; hence, V is the greatest lower bound of S and T.
(join) $W=\wedge_i(S_i\vee T_i)\geq(\wedge_i S_i)\vee(\wedge_i T_i)=S\vee T$. If there is a simple concept X such that $S\leq X$ and $T\leq X$, Theorem 3 states that for all i, $S_i\leq X_i$ and $T_i\leq X_i$, hence $(S_i\vee T_i)\leq X_i$. Therefore, $W=\wedge_i(S_i\vee T_i)\leq\wedge_i X_i=X$, and W is the least upper bound of S and T.
Assume there is a k such that $W_k<(S_k\vee T_k)$, i.e. $v\in P_k$ and $v\wedge W=0$ while $v\wedge S\neq 0$ or $v\wedge T\neq 0$; but then $v\wedge(S\vee T)\neq 0$, contradicting $(S\vee T)\leq W$.
If $S\wedge T=0$, then for all i $V_i=0$, but there may be a k for which $S_k\wedge T_k\neq 0$. ■

THEOREM 5. *Let $\mathcal{P}$ be an environment containing at least two properties, each with at least two values.*

- *(a) SC is atomic.*
- *(b) SC is complemented, although not uniquely.*
- *(c) SC is non-modular.*
- *(d) SC is not semi-modular.*
- *(e) SC is not distributive.*
- *(f) SC is not relatively complemented.*

Proof. (a) Obvious: SC is finite.
(b) SC is complemented iff for any simple concept S, there is a simple concept T such that (*i*) $S\otimes T=0$ and (*ii*) $S\oplus T=1$. (*i*) is satisfied if there is a k such that $S_k\wedge T_k=0$; (*ii*) requires that for all i, $S_i\vee T_i=1$. For any S, there is more than one way to choose T.
(c) Let p and q be two different values of one property, and let r and s be two different values of another property, then SC contains the following 'pentagonal' sublattice:

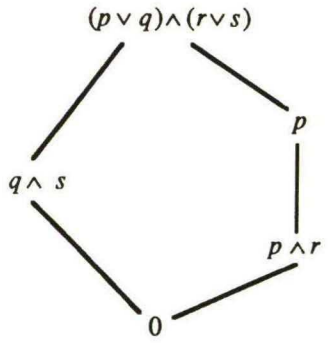


Figure 2. A non-modular pentagonal sublattice

(d) In the pentagonal sublattice of fig. 2, there does not exist an element X with $0<X<q\wedge s$, which is a necessary condition for semi-modularity of a non-modular lattice.
(e) and (f) Immediate from (c). ■

Note, that if the only pentagonal sublattices of SC are like the one in fig. 2, containing 0, then SC is dually semi-modular: just add the elements s and $p\wedge s$.

[11] In fact, this is enough proof for the first part of the Theorem, together with Theorem 3.

Just to achieve some degree of completeness, we consider covers, chains, and lengths of intervals. The covering relation is easily stated in terms of projections.

THEOREM. *In any environment, a simple concept S covers a simple concept T iff either:*

(i) $T=0$, *and for all* i, $S_i \in P_i$;

(ii) $T \neq 0$, *and for some* k, $S_k - T_k = v \in P_k$, *and for all* $i \neq k$, $S_i = T_i$.

Proof. (*i*) Every projection of S equals a value of a property, hence $S>0=T$. Now suppose $S \geq V \geq T$, then for all i $S_i \geq V_i \geq T_i$; if for some k $V_k=T_k=0$, then $V=T=0$; otherwise, because each S_i consists of one single value, for all i $V_i=S_i$, and thus $V=S$. Thus, S covers T.
(*ii*) According to Theorem 3, $S>T$. Suppose $S \geq V \geq T$, then for all i $S_i \geq V_i \geq T_i$; hence, for all $i \neq k$ $S_i=V_i=T_i$, and either $V_k=T_k$ or $V_k=S_k$, implying either $V=T$ or $V=S$. Thus, S covers T. ■

The atoms of the lattice are the covers of 0 as specified in the above Theorem (*i*). Because the lattice is non-modular, it is not true that every simple concept is the join of a **fixed** number of atoms. There is a notion of height, however, expressed by the Jordan-Dedekind condition, stating that for each pair of simple concepts S and T with $S \leq T$, (*i*) all chains from S to T are finite, and (*ii*) all maximal chains from S to T are of the same length.

THEOREM. *In any environment, SC satisfies the Jordan-Dedekind condition.*
Proof. (*i*) *SC* is finite.
(*ii*) The height of a simple concept can be defined inductively as follows: $h(0)=0$, and $h(S)=p-n+1$, where p is the total number of property values in the minimal decomposition of S, and n is the number of properties. We proceed by proving that all maximal chains from S to T are of length $h(S)-h(T)$, and this is established by proving that S covers T only if $h(S)=h(T)+1$.
(a) Let $T=0$. For any atom A, the number of values in its minimal decomposition is n, hence $h(A)=1=h(0)+1$;
(b) If $T>0$, $h(S)=h(T)+1$ follows immediately from the preceding Theorem (*ii*). ■

A.3 Mapping simple terms to simple concepts

THEOREM 6. *The meaning function m is an order quasi-isomorphism. In particular, if ST3 is the unification of ST1 and ST2, then* $m(ST1) \oplus m(ST2)=m(ST3)$; *and if ST4 is the antification of ST1 and ST2, then* $m(ST1) \otimes m(ST2)=m(ST4)$.
Sketch of proof. Requirement (*i*) and (*ii*) are satisfied because of the isomorphism between *ST* and the direct product of the simple disjunctive concept algebras. Requirement (*iii*) is not satisfied, because 0 has several representations. ■

COROLLARY 7. *If ST1 is a correct representation for m(ST1), and ST2 is a correct representation for m(ST2), then the unification of ST1 and ST2 is a correct representation of* $m(ST1) \oplus m(ST2)$, *but the antification of ST1 and ST2 is possibly not a correct representation of* $m(ST1) \otimes m(ST2)$.
Proof. This follows directly from the definition of m, and the second part of Theorem 4. ■

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS