

Pragmatic Data Mining

by Hans van Thiel

"It is the habit of all loose thinkers, to greatly exaggerate the importance of what they have found."
The 19th Century American Philosopher and Logician Charles Sanders Peirce

Version 1.1, August 2006

Summary

This paper describes an algorithm to extract the distinguishing features from sets of rules and group these into new rules. The first part is an introductory overview with some test results, the second part a mathematical description with proofs, and the third lists a 'prototype' implementation in the functional programming language Haskell.

Key terms: categorical data analysis, multivariate data analysis, rule based knowledge, data mining, pragmatism

Author Information:

E: hthiel.char@zonnet.nl
info@j-van-thiel.speedlinq.nl
T: +31 20 6247137
Passeerdersstraat 76
1016 XZ Amsterdam
The Netherlands

Introduction

Sometimes knowledge is expressed as a set of rules, of the form: if this and this and this is the case, then that is the case. Another form might be: these and these and these are those.

Empirical experiments or surveys, where the aim is to find those factors that distinguish between different outcomes, are an example.

Knowledge might also be expressed as a table of attribute-value pairs, where the columns list the mutually exclusive values of the attributes, and the rows list conjunctions of values of different attributes. Such a table does not express any rules as it is, but it is easy to form rules by picking an attribute of interest, and checking if the other attribute values predict the presence or absence of the selected property. This is the case if there are no contradictions, i.e. the same conjunction of values does not predict both that property and its absence.

Knowledge might also be expressed as a tree, where the branches could be dictated by different responses to a questionnaire, for example. Again, choosing a predicate of interest will produce predictive rules, provided there are no contradictions.

The problem with rules like this is that the number of combinations of properties could be very high, if

there are more than a few of them. This makes it hard to find those distinguishing combinations which really appear to affect the outcome, and those which appear to be irrelevant.

Methods to deal with this problem are usually classified as 'multivariate data analysis', 'categorical data analysis' and 'data mining'. The first two are (usually) based on statistical tests, while data mining (mainly) uses search strategies. The advantage of statistical tests is that they can handle inconsistency (contradictions) but the disadvantage is that, by nature, they are probabilistic and ignore exceptions. Another disadvantage is that they do not scale well with the number of attributes and attribute values or predicates. Multivariate data analysis, however, also covers quantitative data, which are not well suited for search strategies.

This white paper describes an algorithm that covers a part of the much larger field of data mining and categorical data analysis. It does not cover quantitative data, and ordinal data (e.g. low, medium, high) are treated as nominal (e.g. red, green, blue).

Its input is a set of rules for something, as well as a set of rules for something (or more things) else, that is/are not that thing.

Its output is a possibly different set of rules that contain no predicates that can be omitted, in other words, contain only the distinguishing properties.

In the first part the algorithm is discussed informally, the second part is a logical-mathematical treatment with proofs, and the third part lists a 'prototype' implementation in the functional programming language Haskell.

Part I: Pragmatic Data Mining

A: Hypothesis and Falsification.

Suppose we have one rule, which states that something that is a P, a Q, an R is an A. We know this to be true, but that is all we have. If this is the case, we put as a hypothesis that a P is an A, a Q is an A and an R is an A. So, instead of one rule, we now have three rules, all shorter than the initial one. This is fairly trivial, but if we know nothing else, it is true.

Now suppose we have another rule, but this one for something which is not an A. This rule states that something which is not a P, but is a Q and an R, is not an A.

So obviously our hypothesis that a P is an A still stands, but some things which are Q appear to be A, while other things which are Q appear to be not-A. The hypothesis for R has been falsified too.

The remaining hypothesis states, that if something is a P, it is a Q.

Now suppose we have a third rule for something which is not an A. It could be anything, provided it is exclusive to A. Suppose the third rule states that something, which is a P, but not a Q, and not an R is not an A.

So now our hypothesis that P is an A has also been falsified. In a table:

P	Q	R	A
not P	Q	R	not A
P	not Q	not R	not A

But we can form a new hypotheses, i.e. that something which is both a P and a Q is an A. This does not contradict any of the rules. We also find that anything which is a P and an R is an A.

So, now we have two new rules for A instead of the original one, each shorter than this original.

These new rule can be found automatically by comparing the row for A (omitting the cell with A) with the rows for not A (omitting the column with not-A) and constructing the logical AND of the predicates that do not match. This logical AND is just the intersection of the sets that are denoted by the singleton predicates (properties).

We can add as many rules for not A as we know, match them with the rule for A, and AND the non-matching predicates to the result we already have. Section two shows that the order in which the matches and ANDs are performed does not matter.

Additional knowledge that things that not P and not Q and not R are not A, for example, does not change the result we got without this rule. If we had known, however, that P and Q and not R imply not A, then this would falsify the rule that the P and Q are A, but leave the one that P and R are A.

In addition to testing for compliance with the rules for not-A, a hypothesis must also be tested for internal consistency. A rule which contains P and not-P, for example, denotes the empty set and must be rejected as a hypothesis. A simple way to check if a new rule is valid, is to check whether its predicates are all in a rule for A. In the prototype implementation described in part III, this is done through a final verification step at the end (also see C on scope and limits).

Of course we can also compare another rule for the property A with the rules we have for not A. It turns out, however, that we do not have to do this, but instead can match all rules for A in one step with each separate rule for not A.

Intuitively this is so, because each match of an A rule consists of the hypothesis that each non-matching predicate, by itself, will predict A. But then we might as well state this hypothesis for all individual predicates at once, and see what combinations the comparisons with the rules for not A turn up.

The matching predicates of a comparison denote the 'universe of discourse' or 'common context'. In the example given, comparing with the first row shows that predicate R is in the rule for A as well as in the one for not A. This means that everything covered by our rules, thus far, is an R.

But if everything is an R, then being an R conveys no information and certainly R is not a distinguishing factor. Therefore it can be left out (but see the next section on scope and limits).

Predicate R cannot be left out in the next comparison, and after comparing with the other rule (for not-A) as well, we find that now there is no explicit 'common context' and we get a rule with R.

B: A Test Result

As an example the following table, originally introduced by J.R. Quinlan, was taken:

Weather	Temperature	Humidity	Wind	Fishing
sunny	hot	high	no	bad
sunny	hot	high	yes	bad
cloudy	hot	high	no	good
rain	mild	high	no	good
rain	cool	normal	no	good
rain	cool	normal	yes	bad
cloudy	cool	normal	yes	good
sunny	mild	high	no	bad
sunny	cool	normal	no	good
rain	mild	normal	no	good
sunny	mild	normal	yes	good
cloudy	mild	high	yes	good
cloudy	hot	normal	no	good

Weather	Temperature	Humidity	Wind	Fishing
rain	mild	high	yes	bad

This data set is complicated enough that the results are not easy to find, yet simple enough that they can be checked by looking. The test results were obtained by running main (see part III, B.10) twice, first with the antecedents for good as A and bad as not-A, and then the other way round. (The sort was done manually.)

cloudy => good

rain no => good
rain yes => bad

sunny cool => good
sunny hot => bad

sunny normal => good
sunny high => bad

sunny mild yes => good
sunny mild no => bad

normal mild => good
normal hot => good
normal no => good
cool no => good
hot yes => bad

These results appear to meet the objective, which was to find the shortest rules that distinguish between selected results. They are all correct, but it has not been tested yet (through a brute force search) whether these are really all, or whether possible predicate combinations have been missed.

C: Scope and Limits

The algorithm is very general, in that it imposes no limits on the rules, except that they do not contradict each other. In particular, the rules may be of different length and the antecedents do not have to be exclusive. (The sets denoted by the antecedents do not have to be blocks of a partition.)

But suppose we have that P and Q are A, and not-P and Q are not-A. Then if we match the first against the second, we get that P implies A. If we match the second against the first, we get that not-P is not-A. This is the desired result, Q is the universe of discourse (common context) and it is irrelevant. But if we now match 'P and Q is A' with the reduced rule that 'not Q is not A' we get both that 'P is A' and 'Q is A'.

This is because the algorithm, or at least the current implementation, assumes that the absence of a

predicate in a rule is equivalent to the presence of its negation (more precisely, something that is a part of its negation). If this is not the case, the result will be incorrect.

If the initial rules are all rows in a table of attribute values (or can be expressed that way) the problem does not occur. Because all the columns have content, if an attribute value is absent, that cell will contain another value of the same attribute, which excludes the absent one.

But we cannot unthinkingly match against the new rules with the distinguishing factors only, without being aware of the common context.

Many real world data consist of responses, which have been based on earlier responses. For example, if a person owns a car, the next question in the survey could be about the price category, while if the person does not own a car, this would not be asked.

Whether the algorithm is suitable, or can easily be adapted, for this type of rules is unclear at present.

A related issue concerns hypotheses which are not supported by the original rules for A. These denote the empty set and some should be rejected right away, like windy and not windy, for example. Another example, however, from the above test, was 'hot and rainy' for both good and bad fishing. This combination does not occur in the table and so it also denotes the empty set and it should be rejected. However, it is not self contradictory like 'yes and no' for 'wind' and maybe it could be treated as a suggestion that potentially useful information is missing in the original table. The current prototype, however, just removes all rules which appear to denote the empty set.

D: Some Remarks on Interpretation

The proof in part two uses well founded first order predicate logic and general set theory. The idea, however, was strongly influenced by the pragmatism of the 19th century American philosopher and logician Charles Sanders Peirce. (Peirce was also the one who 'invented' induction and abduction as empirical counterparts of syllogistic deduction.)

In first-order predicate logic, a predicate is treated as a descriptive entity. In this logic an implication is of the form, if an x has a predicate (or combination of predicates) then it also has another (combination of) predicate(s). Because the implication of something that is false is always true, by definition), we can quantify over any number of things x.

Peirce called this use of predicates 'iconic' , but pointed out another use, i.e. as indices. In this theory the antecedent of an implication is the 'index', and the consequent the 'icon'. So, the predicates of the index denote the 'universe of discourse', that which we are talking about.

In a pragmatic interpretation, then, the algorithm described here enhances the 'universe of discourse' by shortening the rules as well as by splitting them. The first is the case because fewer predicates denote larger (or at most equal) sets. Splitting transforms a set intersection into a union, which is also larger, or at most equal. So, the 'indices' are now broader, we have increased the domain of quantification.

The method is also fundamentally empirical and incremental. We have some rules for something, based on this we formulate a hypothesis, and then we try to falsify this hypothesis and formulate new ones. This could be an ongoing process and we can always adapt our new hypotheses to new rules we might discover later. The only provision is that any rules we might add are not contradictory to those

we already have, and not self-contradictory. This flexibility might be useful for for data warehousing applications and monitoring in general.

The principle of empirical adaption also fits very well into the pragmatic philosophy of science and knowledge, of which Peirce is the founding father.

Part II: Reducing Logical Implications

A: The Problem

A logical implication is a conjunction of predicates, which together imply another predicate. In mathematical notation:

$$\forall x(P(x) \wedge Q(x) \wedge R(x) \wedge \dots \Rightarrow A(x))$$

This might possibly be reduced in two ways. Firstly, predicates might be superfluous for the determination of the consequent and, secondly, there might be more of such combinations. For example:

$$\forall x(P(x) \wedge Q(x) \Rightarrow A(x))$$

$$\forall x(P(x) \wedge R(x) \Rightarrow A(x))$$

The problem of reducing an implication is to find the lists of shortest implications and to find all of them.

B: Ancillary Propositions:

B.1: Let P and Q denote predicates in the first proposition, and the sets denoted by those predicates in the second. Then:

$$\forall x(P(x) \wedge Q(x) \Rightarrow A(x))$$

and:

$$\forall x((P \cap Q)(x) \Rightarrow A(x))$$

are equivalent by definition.

B.2:The propositions:

$$\forall x(P(x) \Rightarrow A(x)) \wedge \forall x(Q(x) \Rightarrow A(x))$$

$$\forall x((P \cup Q)(x) \Rightarrow A(x))$$

are equivalent. The second proposition states that any element, which is in P or Q or both is also in A. The first proposition states the same.

B.3: Whenever two implications of the form:

$$\forall x (P(x) \wedge Q(x) \Rightarrow A(x))$$

$$\forall x (P(x) \Rightarrow A(x))$$

are found, the first reduces to the second. This is so because if:

$$\forall x (X(x) \Rightarrow A(x))$$

then:

$$\forall x (X(x) \wedge Y(x) \Rightarrow A(x))$$

for any predicate Y, because antecedents with all the predicates of another and some more denote subsets, and set inclusion is transitive. (If all ravens are black, then all red-eyed ravens are black too.)

C: Reduction of implications:

C.1: Reduction of one implication only

Because of B.1 and B.2 we can represent all implications, for the same consequent A, by a union of intersections of predicate symbols.

Suppose we have just one implication:

$$\forall x (P(x) \wedge Q(x) \wedge R(x) \Rightarrow A(x))$$

The reduction is simple but also meaningless, that is, all P are A, as well as all Q and all R. (In a database of ravens only, everything is a raven.)

C.2: Reduction of one implication of A, using one for not-A

To be able to reduce an implication of predicate A, we need implications for predicate not-A.

Let the two implications for A and not-A be:

$$\forall x (M(x) \wedge Y(x) \Rightarrow A(x))$$

and:

$$\forall x (M(x) \wedge Z(x) \Rightarrow \neg(A(x)))$$

in which M(x) is the conjunction of predicates which occur in both.

M may be empty, but Z and Y must each consist of at least one predicate. If not, the predicates in the two antecedents would form subsets and the implications would be inconsistent. (If all ravens are birds then all red-eyed ravens cannot be mammals, which are not birds.)

For just two implications, one for A and one for something which cannot be A, the predicate M actually denotes all elements x over which we quantify, the universe of discourse. Everything, both A and not-A, is an M. But then we can safely leave M out of the reduced implications. (If the only things we are looking at are all cars, the fact that some things are also cars is not a distinguishing feature.)

C.3: Reduction of one implication of A with two for not-A

Now suppose there are two different not-A implications we can use to reduce the A- implication:

$$\forall x(X(x) \Rightarrow A(x))$$

This can then be written in two ways:

$$\forall x(M(x) \wedge Y(x) \Rightarrow A(x))$$

by matching with the first, and:

$$\forall x(N(x) \wedge V(x) \Rightarrow A(x))$$

by matching with the second. N consists of those predicates in both the A and the second not-A implication.

M and N themselves may have predicates in common or not. Let that list be BNM.

$$\forall x(BNM(x) \wedge U(x) \wedge Y(x) \Rightarrow A(x)) \quad (1)$$

$$\forall x(BNM(x) \wedge W(x) \wedge V(x) \Rightarrow A(x)) \quad (2)$$

Reducing against the first not-A implication produces a union Y of predicate symbols and reducing against the second produces a union V. Obviously:

$$U \cup Y = W \cup V$$

but:

$$U \cap W = \emptyset$$

therefore:

$$U \subset V \quad \text{and} \quad W \subset Y \quad \text{or vice versa.}$$

So, if we have only Y and V, we still have all predicates except those in both the N and M matchings. But these BNM predicates, as discussed above, denote the (new) universe of discourse and may safely be left out.

Suppose Y is the union of P and Q, and V is the union of P and R. Then the first reduction states:

$$\forall x(P(x) \Rightarrow A(x))$$

$$\forall x(Q(x) \Rightarrow A(x))$$

and the second:

$$\forall x(P(x) \Rightarrow A(x))$$

$$\forall x(R(x) \Rightarrow A(x))$$

The implication with P still stands, but the one for Q is obviously falsified by the second, since it is not in V, but in W, which is in the matching N. Likewise, the implication for R is falsified because R is in U which is in the matching set M.

Suppose, however, we add R to Y and Q to V, producing Y' and V'. Because the intersection of a set with itself is that set, the two matches (1) and (2) remain the same, but:

$$Y' = (P \cup (Q \cap R)) \quad \text{and} \quad V' = (P \cup (Q \cap R))$$

and Y' is not in M, nor in N. So, a correct result of the two matches is:

$$\forall x (P(x) \Rightarrow A(x))$$

$$\forall x (Q(x) \wedge R(x) \Rightarrow A(x))$$

(If we have that red, fast, cars are nice, black, fast, snakes are ugly, red, slow, lawn-mowers are dull, then cars are nice, also 'red and fast' is nice, but not everything red, nor everything fast.)

In the general case we will have the symbols P and Q in Y, and symbols R and S in V. Because symbols R and S are in U we can substitute P twice, by its intersection with R and with S. Likewise for V, and so we get four new predicates. Of course, in the case where one predicate appears twice:

$$(P \cup Q) \cap (P \cup R) = P \cup (Q \cap R)$$

as in the above example. In general, the reduction of the A-implication is represented as the union of all distinct pairs of intersections.

C4: Reduction of one A-implication with more than two for not-A

In the case of two not-A implications, as discussed in C.3, the predicates in Y', which is equal to V', are the predicates that match neither M nor N. These new predicates are not really new, however, but just re-orderings of predicates which are already there. In fact, both (1) and (2) (and the original A-implication as well) can be written as:

$$\forall x (BNM(x) \wedge Y'(x) \Rightarrow A(x))$$

Suppose we match with a third implication of not-A and find matching predicates L and a non-matching set Z. As before we will find a new universe of discourse BNML, whose predicates can be left out.

The new reductions will be the pair wise intersections of Z and Y'.

Z, Y and V are all obtained by matching the same set of predicates, the antecedent of the A-implication, with antecedents of the not-A implications, so the order in which they are performed is irrelevant.

The resulting universe of discourse BNML is also independent of the order in which the matches are executed. Y' is just the intersection of Y and V, and because intersection is both commutative and

associative, their ordering does not matter either.

So the algorithm for reduction consists of matching the antecedent for A with all those for not-A, and intersect all the unions of non-matching predicate symbols.

When constructing the pair-wise intersections we can use:

$$P \cap (P \cap Q) = P \cap Q$$

and:

$$P \cup (P \cap Q) = P$$

for any predicate (or combination of predicates) P and Q.

C.5: Reduction of more than one implication of A with more of not-A

Obviously there could be more than one implication for A, as well as for not-A, and we can reduce each of these implications with all the implications for not-A. Suppose two of these reductions are:

$$\forall x (B(x) \Rightarrow A(x))$$

and:

$$\forall x (C(x) \Rightarrow A(x))$$

Because of B.3, one of these implications can be reduced to the other, if B is a sub set of C, or vice versa.

So, we can further reduce a list of A-reductions by applying:

$$P \cup (P \cap Q) = P$$

for any reductions which are obtained from different implications of A. (This has already been done for the results from each separate implication of A.)

C.6: Reduction of more than one implication of A in one step

Suppose we have two different implications of A and we match with two implications of not-A. Let the matches be E1 and F2 from the first of the A implication, and G1 and H2 for the second A-implication. The reduction will then be:

$$(E1 \cap F2) \cup (G1 \cap H2)$$

which can be reduced if it is equal to one of its components. Let:

$$(E1 \cap F2) \cup (G1 \cap H2) = (G1 \cap H2)$$

and therefore:

$$(E1 \cup G1) \cap (E1 \cup H2) \cap (F2 \cup G1) \cap (F2 \cup H2) = G1 \cap H2$$

There is no direct way to check this, without checking the intersections themselves.

We can, however, leave out any set of non-matching predicate symbols if it has to be joined with a larger one, using:

$$P \cup (P \cup Q) = P \cup Q$$

Let the two predicate symbol sets for A be X and X', and the two for not-A be O and T. Then E1, the non-matching predicates with O, is simply the set difference between X and O. G1 is the difference between X' and O.

But for any sets in general:

$$(X \setminus O) \cup (X' \setminus O) = (X \cup X') \setminus O$$

and we can directly match the predicates in both X and X' with O. The same applies to the union of F2 and H2, and we get:

$$(X \setminus T) \cup (X' \setminus T) = (X \cup X') \setminus T$$

With E1 the difference of X with O, and H2 being X' without T:

$$(X \setminus O) \cup (X' \setminus T) = (X \cup X') \setminus (O \cap T)$$

With F2 the difference of X with T, and G1 being X' without O:

$$(X \setminus T) \cup (X' \setminus O) = (X \cup X') \setminus (O \cap T)$$

and so these cases are identical.

But also:

$$((X \cup X') \setminus O) \cap ((X \cup X') \setminus T) = (X \cup X') \setminus (O \cap T)$$

so these two (identical) cases are covered when we match the union of X and X' (i.e., all the predicate symbols in either one or both) with the the antecedents O and T of the implications of not-A.

If this is true for two implications of A and two of not-A, then obviously it is true for more than two implications of A and two of not-A. The rest of the reduction algorithm is the same as for one A-implication and more than two implications of not-A, so the method may be applied to any arbitrary number of implications.

Therefore, while we cannot quickly determine whether an implication of A may turn out to be covered by the reductions of other implications of A, we can reduce all of them in one step, by taking the union of their predicate sets, matching those against the available implications of not-A and intersecting the resulting unions.

Part III: A 'Prototype' Implementation in Haskell

A: Introduction

Haskell is a functional programming language (see www.haskell.org) which is very suitable for prototyping and testing, among others, because Haskell programs are short and close to the methods they implement.

The following is a listing and documentation of a Haskell implementation of the algorithm for reducing logical implications. This algorithm is discussed in the other sections of this paper.

The individual predicates may be represented by any type for which equality is defined, e.g. strings, indexes into attribute-value lists, etc. In module Main (see B.10) the general type in class Eq of module Peirce (see B.1 - B.9) is treated as a String type.

An antecedent is a list of predicates and represents the intersection of the underlying sets.

A union of antecedents (for the same consequent) is represented as a list of lists.

For efficiency, however, a union of singleton predicates is also represented by a single list, instead of by a list of singleton lists, until the last moment.

All functions have been tested with the Hugs 98 Haskell interpreter, running on a Fedora 2 version of Linux. (Hugs is also freely available for Windows.)

This implementation uses no other Haskell modules apart from the Prelude.

B: The Haskell Functions

B.1: The first step in a reduction is to get the predicates in an antecedent for A, which are not in an antecedent for not-A. Only the antecedents of implications are matched, they may not contain any consequents.

```
-- The value of (xnomatchy x y) is a list of all elements
-- in the first list x, which are not in the second list y
```

```
xnomatchy :: Eq a => [a] -> [a] -> [a]
xnomatchy [] y = []
xnomatchy (x:xs) y =
    if x `notElem` y then x: xnomatchy xs y
    else xnomatchy xs y
```

```
-- Note: there is no test for doubles in x and they are not removed
```

B.2: If there are more implications of A, their antecedents will all be all matched with an antecedent (of an implication for not-A) in one step.

```
-- the value of antsnomatchy is a list of all elements
-- in a list of lists which are not in a list y
```

```

antsnomatchy :: Eq a => [[a]] -> [a] -> [a]
antsnomatchy [] y = []
antsnomatchy (x:xs) y =
    joinxyls (xnomatchy x y) (antsnomatchy xs y)
    where
        joinxyls x y = (xnomatchy x y) ++ y

```

B3: When two antecedents for not-A have been matched with the A-antecedents, the next step is to construct their intersections, in other words, to repeatedly AND the predicates pair wise. For one predicate, and one AND list of predicates, this means adding it to the list, except if it is in the list already. The function andtolists uses andtols to do this for a list of lists, and andlists then ANDs a list to a list of lists.

```

-- andlists produces the logical 'and' of
-- a list with a list of lists

andlists :: Eq a => [a] -> [[a]] -> [[a]]
andlists [] y = []
andlists (x: xs) y = (andtolists x y) ++ (andlists xs y)

-- andtols produces the logical 'and' of a
-- (singleton) value and the lists in a list.

andtolists :: Eq a => a -> [[a]] -> [[a]]
andtolists x [] = []
andtolists x (y:ys) =
    (andtols x y) : (andtolists x ys) where
        andtols x y = if elem x y then y
                      else x:y

```

B4: In general the value of andlists will contain super sets of predicates and/or doubles, which must be removed.

The function weqredands produces a list of lists in which there are no more super sets. It uses abottom to find the minimum for each list, which in its turn uses minifls to check whether the first list is a sub set of the second. The function abottom is then mapped onto the list (of lists).

```

-- weqredands reduces a list of 'and lists'
-- but leaves all doubles (set wise)

-- weqredands reduces a list of 'and lists'
-- but leaves all doubles (set wise)

weqredands :: Eq a => [[a]] -> [[a]]
weqredands x = map ((flip abottom) x) x

-- abottom produces a bottom of a list in
-- a list of lists (there may be more)

abottom :: Eq a => [a] -> [[a]] -> [a]
abottom x [] = x
abottom x (y:ys) = abottom (minifls x y) ys
    where minifls x y =
            if xnomatchy y x == []
            then y

```

```
else x
```

The function `remdub` removes the doubles from a list of lists and uses the comparison function `seteq`. For each list it looks if it has doubles, and only takes those lists which do not. For this it uses a general function `setpred`, which compares all elements in a list.

```
-- seteq is True if two lists have the same elements
-- NOTE: disregards doubles in x or y, i.e. [1] == [1,1]
```

```
seteq :: Eq a => [a] -> [a] -> Bool
seteq x y = if (xnomatchy x y == []) &&
              (xnomatchy y x == [])
              then True
              else False
```

```
-- setpred is True if a list satisfies a
-- property with elements of a list of lists
```

```
setpred ::
  Eq a => ([a] -> [a] -> Bool) -> [a] -> [[a]] -> Bool
setpred pred x [] = False
setpred pred x (y:ys) = if pred x y then True
                        else setpred pred x ys
```

```
-----
-- remdub removes the doubles from a list of lists
```

```
remdub :: Eq a => [[a]] -> [[a]]
remdub [] = []
remdub (x:xs) = if not (setpred seteq x xs)
                then x: remdub xs
                else remdub xs
```

The function `redands` now produces a list of and lists in which there are no super sets and no doubles.

```
-- redands reduces a list of 'and' lists
```

```
redands :: Eq a => [[a]] -> [[a]]
redands x = remdub (weqredands x)
```

B.5: To reduce a list of antecedents of A-implications with a list of antecedents of not-A implications we can use the functions defined before in a function `redimpswot` (`wot` stands for, without test, see B.6). Because the ANDs are performed on a list of lists, the first match needs to be converted to that format with a function `raisetolist`.

```
-- help function to initiate redimpswot
```

```
raisetolist :: [a] -> [[a]]
raisetolist [] = []
raisetolist (z: zs) = [z] : (raisetolist zs)
```

```
-----
-- redimpswot reduces a list of antecedents in an
-- implication by checking the negation antecedents
-- Note: the first list is the one which is reduced.
```

```
redimpswot :: Eq a => [[a]] -> [[a]] -> [[a]]
```

```

redimpswot x (y:[]) = raisetolist (antsnomatchy x y)
redimpswot x (y:ys) =
  redands (andlists (antsnomatchy x y) (redimpswot x ys))
-----

```

B.6: It might be useful to test the data set for consistency first. If a list in the A-list is a sub set of a list in the not-A list, then we have the case where something implies both A and something which is not A. The function setsub tests for this, and consistent then uses this in the function setpred (see B.4) to do the test.

```

-- setsub is True if in two lists one contains all
-- elements of the other. Note: see seteq

setsub :: Eq a => [a] -> [a] -> Bool
setsub x y = if (xnomatchy x y == []) ||
               (xnomatchy y x == [])
               then True
               else False

-- consistent tests the data set for consistency
-- implication and negation antecedents no sub sets

consistent :: Eq a => [[a]] -> [[a]] -> Bool
consistent [] y = True
consistent (x:xs) y = if (setpred setsub x y)
                       then False
                       else consistent xs y

-- redimps reduces the antecedent implications
-- after checking consistency with the negation

```

B.7: The function which reduces a list of A-implications using a list of not-A implications, which also checks for consistency, is redimps.

```

redimps :: Eq a => [[a]] -> [[a]] -> [[a]]
redimps x y = if consistent x y
               then redimpswot x y
               else error "Inconsistency in Data"

```

B.8: The functions redimpswot and redimps will, however, also produce hypotheses that are not supported by the implications of A, in particular combinations of predicates that denote the empty set. Examples are different values of the same attribute from a table. To correct this the results can be compared to the A implications. The function verify filters all lists which are not sub lists of an A-antecedent.

```

-----
-- xisuby is true if x is a sub set of y

xisuby :: Eq a => [a] -> [a] -> Bool
xisuby x y = if xnomatchy x y == [] then True
              else False
-----

-- verify produces the lists in the first list (result)
-- that are sub sets of a list in the second (A anteced)

verify :: Eq a => [[a]] -> [[a]] -> [[a]]

```

```

verify [] y = []
verify (x:xs) y = if setpred xisubby x y
                  then x : verify xs y
                  else verify xs y
-----

```

B.9: The function `vredimps` is the verified version of `redimps`.

```
-- vredimps is redimps with non sub sets of A removed
```

```
vredimps :: Eq a => [[a]] -> [[a]] -> [[a]]
vredimps x y = verify (redimps x y) x
```

B.10: The following is an implementation of `vredimps` where `a` is of type `String`. The functions described above are in module `Peirce` (file `Peirce.hs`). The antecedents are lines of words separated by white spaces. Those for `A` are read from the file `a_rules.txt`, those for `not-A` from file `nta_rules.txt`. The function `streduce` performs function `vredimps` on lists produced by `str2ants` and converts the result back to a string (through `ants2str`), which is written to the file `red_rules.txt`.

```

module Main
  where

import IO
import Peirce

main = do
  astr <- readFile "a_rules.txt"
  nastr <- readFile "nta_rules.txt"
  writeFile "red_rules.txt" (streduce astr nastr)

str2ants :: String -> [[String]]
str2ants x = map words (lines x)

ants2str :: [[String]] -> String
ants2str x = unlines (map unwords x)

streduce :: String -> String -> String
streduce s1 s2 =
  ants2str (vredimps (str2ants s1) (str2ants s2))

```

Version History

Version 1.0	July 2006	first version for publication
Version 1.1	August 2006	added verify and main to part III, the test result to part I and clarified part I