Paper presented at Expert Systems '90, the Tenth Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems, London, September, 1990. In T R Addis and R M Muir (Eds.), *Research and Development in Expert Systems VII*, Cambridge: Cambridge University Press, 1990, pp. 263-276.

UNIFYING COMPUTING: INDUCTIVE LEARNING AND LOGIC

J Gerard Wolff

Andrew J Chipperfield

School of Electronic Engineering Science, University of Wales, Dean Street, Bangor, Gwynedd, LL57 1UT, UK. Telephone: +44 248 351151 ext 2691. Electronic mail: j.g.wolff@uk.ac.bangor.sees.

Abstract

The paper introduces a new theory of computing, called 'SP', and describes a prototype of a new kind of computing system (also called SP) which is based on the theory. The focus of the paper is on how the SP theory and the SP system may integrate two kinds of computing which are not normally related: inductive learning and logic.

The SP theory is based on the conjecture that all kinds of computing and formal reasoning may usefully be seen as a search for *efficiency* in information, where the concept of efficiency is defined in terms of the concept of *redundancy* in Shannon's information theory.

The SP prototype (SP6) provides a simple language intended for the representation of diverse kinds of knowledge. The prototype is dedicated to searching for redundancy in knowledge and extracting it wherever it is found. Elements of this process are pattern matching and unification of patterns combined with a hill-climbing search amongst alternative sets of unifications.

The paper describes examples showing how SP6 performs the induction of a grammar from examples, the induction of a class hierarchy from examples, simple syllogistic reasoning and logical reasoning with a transitive relation ('equals'). Issues for future research are discussed.

1. Introduction

What has inductive learning got to do with logic? "Not much" is the answer implied by most research in these two areas. The two topics are not normally connected.

This paper introduces a theory of computing within which learning and logic may be integrated. The theory is relevant to several other areas of computing - including probabilistic inference, information retrieval, natural language processing, pattern recognition, object-oriented design, problem solving and automatic planning - but these topics will not be considered here. The focus in this paper is on how inductive learning and logic may be seen as two facets of one computing process.

In the paper, we first give an outline of the theory, called 'SP'. Then we describe a prototype of a new kind of computing system - also called SP - which is based on the theory. Last, we describe and discuss examples from the prototype showing how inductive learning and logical inference may both result from one unified computing process.

2. The SP Theory

The central idea in the SP theory (described in Wolff (in press) and Wolff (1989)) is the conjecture that all kinds of computing and formal reasoning may usefully be seen as a search for *efficiency* in information, where the concept of efficiency is defined in terms of the concept of *redundancy* in Shannon's information theory.

In the SP theory, the efficiency of a body of information (I) is defined as:

$$efficiency = power / size$$

Power is defined as the information content of I (in bits), **excluding** any redundancy in I.

Size (otherwise known as *simplicity*) is defined as the information content of I (in bits), **including** any redundancy in I.

2.1. The SP Search Space

The SP search space is illustrated in figure 1 below.

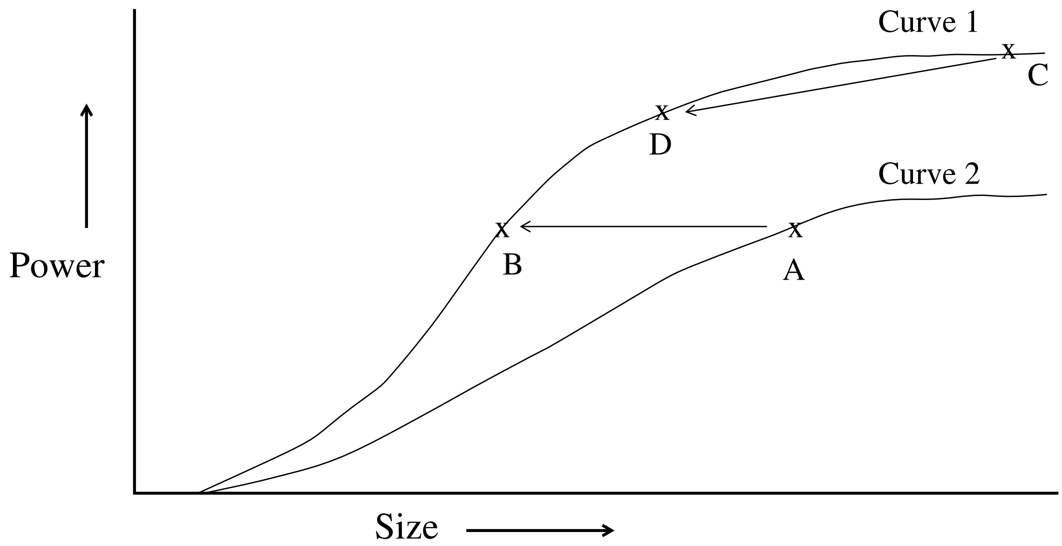

Figure 1. The SP Search Space

Each point in figure 1 represents a body of information or a knowledge structure with its own measure of size and power.

Each curve in the figure illustrates the trade-off between size and power: big knowledge structures are usually more powerful than little ones and *vice versa*.

Curve 1 represents a set of relatively efficient knowledge structures at varying levels of size and power while curve 2 represents relatively inefficient knowledge structures varying in a similar way.

2.2. Pattern Matching, Unification and Hill-Climbing Search

Searching for efficiency in a body of information means, in large part, searching for redundancy in the information and extracting the redundancy wherever it is found:

1. Redundancy means repetition of information. Where a pattern of information repeats more often than other patterns of the same size, there is redundancy.

   The elements of a 'pattern' are not necessarily contiguous. In many bodies of information, especially natural language, a significant part of the redundancy is due to constraints and contingencies which bridge other structures.

2. Redundancy in a body of information may be detected by comparing or *matching* patterns.

3. Redundancy in a body of information may be reduced by the merging or *unification* of repeating instances of a pattern to make just one.

4. The complication is that there are often many alternative ways in which a given pattern or part of a pattern may be unified. Finding a good set of unifications for the repeating patterns in a body of information amongst all the many possible sets of unifications means a process of *searching*. Because the search space is usually very large, often infinite, it is necessary to use a *hill-climbing* technique - or something equivalent - to achieve an effective search in a reasonable time.

To find and extract as much redundancy as possible means searching for the largest possible value of **R** where:

$$R = \sum_{i=1}^{i=n} (f_i - 1) \cdot s_i ,$$

f is the frequency of the **i**th member of a set of **n** patterns and **s** is its size in bits.

2.3. Variations

The process of searching for efficiency in information can take a variety of forms:

- **Preserving power or sacrificing power**. The search may be constrained to preserve all the power in the body of information I. In the diagram, a transformation in a knowledge structure which moves it from A to B means increasing its efficiency without reducing its power. If the circumstances are right, the knowledge structure may be moved 'down market' by sacrificing power in order to gain an overall increase in efficiency. In the diagram, a transformation in a knowledge structure which moves it from C to D reduces its power but increases the ratio of power to size and thus increases efficiency.

- **Breadth of search**. Small knowledge structures may be searched exhaustively. As already indicated, most realistically large knowledge structures cannot be searched exhaustively - but the thoroughness of the search may vary. *Broad* searching is hungry for computing resources and is likely to need high levels of parallelism in processing to achieve adequate speed. The benefit of broad or thorough search in a computing system is that the system can be relatively flexible and intelligent. *Narrow* searching reduces computational costs but means that the computing system has less flexibility and more of that 'brittleness' and stupidity which characterises current systems.

- **One path or multiple paths**. The search process may thread one path through the search space and give one 'best' answer. Alternatively, the search process may follow two or more paths through the search space and give a set of answers, graded in terms of the efficiency they represent. These parallel paths may be followed in sequence - in which case we have the effect of 'backtracking'. If there are sufficient resources, they may be followed in parallel.

2.4. Related Work

The SP theory originated in a programme of work on language learning by children, with research in inductive learning of grammars as a major part of the programme (Wolff, 1987, 1988). The similarity between the processes needed for the inductive learning of grammars and processes at the heart of resolution theorem proving is what lead to the SP theory.

The idea of searching for economical structures, which is a central part of the SP theory, features in some research on 'neural' computing (see, for example, Campbell, Sherrington & Wong, 1989, and Hinton & Sejnowski, 1986) and also in research on cluster analysis and numerical taxonomy. The SP theory presents a view of computing which is not tied to any particular architecture (such as neural networks) and which is in several other respects different from work on neural computing or cluster analysis. There is potential in the theory to integrate the 'connectionist' and 'symbolic' views of computing.

The idea that all kinds of formal system and computing system may be seen as information has been developed as 'algorithmic information theory' (see, for example, Chaitin,1987). That work focuses on issues of completeness and decidability in formal systems whereas the SP theory focuses on how an analysis of formal systems in terms of information and redundancy may illuminate issues in knowledge representation and the process of computation itself.

The SP language (described below) owes its greatest debt to Prolog (see, for example, Clocksin & Mellish, 1981) but differs from Prolog in several ways. There is some resemblance between SP and 'unification grammars' (see, for example, Shieber, 1986) but also significant differences. Unification concepts are reviewed in Siekmann (1989). The most important difference between SP and these kinds of system is that the pattern matching and unification processes in SP are much more general and they are set within a broader theoretical frame. This generalisation gives SP capabilities - like an ability to learn - which are not exhibited by Prolog or by systems of that type.

SP incorporates many of the concepts associated with object-oriented design (reviewed in Cook, 1986).

Although extraction of redundancy in information is a key part of the SP theory, the SP system is **not** a *reduction* system in the technical sense of that word (see, for example, Robinson, 1988).

Although SP can imitate the effect of a re-write system it does **not** use any re-writing technique in the technical sense of that term (see, for example, Beaten, Bergstra & Klop, 1987).

The SP system may be loosely classified as a 'knowledge manipulation engine' like the FACT database system (McGregor, McInnes & Henning, 1987), the

‘Intelligent File Store’ (Lavington, 1988) or the integrated database and expert system (Syntel) described by Risch, Reboh, Hart & Duda, 1988) but there are many differences between SP and these systems in orientation and expected capabilities.

3. SP6

SP6 is a software simulation of SP, written in C, which runs on a Sun workstation. The following subsections describe the elements of this prototype and some of its internal workings.

3.1. The SP Language

In the SP system, all kinds of information or knowledge - ‘data’, ‘programs’, ‘results’ - are expressed in one simple language.

A formal syntax for this language is given in Wolff (in press) and repeated (with small changes) here:

```
Object --> Ordered-AND-object | Unordered-AND-object
        OR-object | Simple-object
Ordered-AND-object --> ‘(’ , body , ‘)’ ;
Unordered-AND-object --> ‘[’ , body , ‘]’ ;
OR-object --> ‘{’ , body , ‘}’ ;
body --> (Object, body) | NULL ;
Simple-object --> symbol | ‘_’ ;
symbol --> (character , symbol) | character ;
character --> ‘a’ | ‘A’ | ... | ‘z’ | ‘Z’ | ‘0’ | ... | ‘9’ | ‘%’ ;
```

Less formally, any item of information or knowledge in the SP language is represented as an *object*. SP objects come in four main types:

- **(...)**. Zero or more objects enclosed in round brackets are an *Ordered AND Object (OAO)*. In an OAO, the order or sequence of the constituent objects is significant.

  At some stage in the research programme, it is likely that the OAO construct will be generalised to two- or three-dimensional structures as well as simple sequences. This will mean that the SP language can be used to represent such things as maps or three-dimensional models of real world objects.

- **[...]**. Zero or more objects in square brackets are an *Unordered AND Object (UAO)*. In a UAO, the order or sequence of the constituent objects is not significant. The concept of a UAO is similar to the concept of a ‘set’ in mathematics or, more accurately, a ‘bag’ because two or more of the constituents of a UAO may be identical.

- **{ ... }**. Zero or more objects enclosed in curly brackets are an *OR Object (ORO)*. In an ORO, the constituent objects are treated as alternatives in the context in which the ORO is found.

- *Simple objects* come in two sub-types:

  - A *symbol* is one or more alphanumeric characters. If the symbols are always one fixed length, boundary markers are not needed. In current prototypes, the symbols can vary in length and boundary markers (spaces and commas) are used.

- A *variable*, represented by an underscore ('_'), functions as a place marker for information which is missing from a structure. It is similar in some ways to the concept of an unnamed variable in Prolog. There is some discussion, below, of what the SP variable means.

Unlike Prolog, and most systems of that kind, SP does not have named variables. The effect of naming can be achieved in SP by bracketing a 'name' with an object to be named inside a UAO or OAO, eg [Fred _]. Normally, a name will be a SP symbol but other kinds of objects may be used as names.

Every symbol in an SP object has a frequency value which shows how many identical copies of that symbol were unified to create the given symbol. In the examples presented later, frequency values are not shown in full because this makes the examples difficult to read. Instead, symbols with a frequency of 1 are shown in plain type and symbols with any frequency greater than 1 are shown in bold type.

Sets of symbols used in versions of the SP language may vary from large sets of variable-length character strings down to the familiar binary set of single characters, 0 and 1. The symbol set used in any given SP object controls the 'granularity' of the operations in SP.

3.2. Processing in SP6

The 'search' function provides the semantics of the SP language and expresses the SP theory of computing, as currently understood. In terms of the options described earlier, the search process in SP6 works like this:

- The search for efficiency in SP6 means searching for redundancy in the corpus and extracting it wherever possible but without removing any non-redundant information. In other words, the search process is designed to preserve all the power in the corpus. Exploring how power may be sacrificed to improve efficiency has not been attempted yet in this research programme.
- The search process is systematic but not totally exhaustive. There are some kinds of redundancy which it cannot detect.
- SP6 produces one 'best' answer rather than a set of alternative answers (which future versions of SP will aim to produce).
- A simple and relatively unsophisticated kind of hill-climbing technique is being used.

3.2.1. Finding Redundancy

SP6 searches for redundancy by searching for patterns which match each other. To do this, it makes systematic comparisons (all-or-nothing matches) between all the constituent objects of the corpus and all sub-sets of the OAOs in the corpus, where the members of each sub-set are contiguous within the OAO from which they are drawn. SP6 does not compare sub-sets of UAOs although it would be relatively easy to extend the search in this way.

Notice that, although the matching process gives a simple all-or-nothing result, **SP6 detects partial matches between objects**. This is because the all-or-nothing matching process is applied to sub-sets of OAOs as well as whole OAOs.

3.2.2. The Unification Process

Where two patterns match, they may be unified. This is achieved in four main ways:

- Where two objects match and are siblings within a UAO or ORO, they may be unified to give an object which replaces them within the parent. For example:

  [A A B] => [**A** B]

  (The symbol '=>' is not part of the SP language. It is used to show how the SP system may transform one SP object into another.)

- Where two objects match and have an 'uncle / nephew' relationship within a UAO (or grand-uncle / grand-nephew relationship, etc.), they may be unified and replaced by an object in the location of the nephew. In effect, the uncle 'moves' to merge with its nephew. Here is a simple example:

  [A (A B) C] => [(**A** B) C]

  The argument here is that if the uncle moves to join the nephew then there is no loss of non-redundant information but if the nephew moves to join the uncle then there is loss of non-redundant information. The uncle is merely 'associated' with the other objects and this association is not destroyed by the unification. But if the nephew moves then there is loss of information about its position relative to other objects: the fact that it precedes B within (A B) in the example.

- Where there is a partial match between two OAOs which are siblings within a UAO, the parts which match are merged to make one pattern and the parts which do not match are formed into an ORO. For example,

  [(A B C D E F) (A B C X Y Z)]
  =>
  (**A B C** {(D E F)(X Y Z)})

  In SP6, there are constraints on when this kind of unification can be done which are designed to ensure that there is no loss of power in the corpus.

- In cases where the other three kinds of unification cannot be applied, SP6 may achieve unification by the creation of 'references' and 'definitions'. A reference is a UAO containing a 'reference tag' and a variable like this: [%1 _]; the corresponding definition is a UAO containing the same reference tag and another object - the 'body' of the definition - which is typically an OAO. The following example illustrates how references and a definition may be used to effect the extraction of redundancy in an object:

  (A B C D E F G H I J K L X Y Z A B C D E F G H I J K L)
  =>
  ([%1 (**A B C D E F G H I J K L**)] X Y Z [%1 _])

  The general idea is to record the repeating pattern once within the body of a definition, leaving references to mark where the original patterns came from. With this mechanism, redundancy may be extracted without losing the non-redundant information about where the original patterns were located within the corpus.

  References and definitions are only created where the redundancy which is extracted is greater than the information 'cost' of the symbols which are introduced by the transformation.

With one qualification, references and definitions do not have any formal status in the SP theory. A reference tag is simply a 'pattern' with the same status as any other pattern. The mechanisms in SP for pattern matching and unification are used to bring the body of a definition back into the context whence it came.

The qualification, just mentioned, is that the SP system knows when it has added reference tags to a structure. Any tags added in this way are distinct from information supplied to the system by the user. For this reason, the system can undo references and definitions which it has created at one stage in the search for efficiency if, at a later stage, they are no longer needed.

3.2.3. Scheduling of Processes

The compress function of SP6 extracts redundancy in an iterative way like this:

```
While no more extractable redundancy in the corpus do
BEGIN
        1 Search for redundancy in the corpus by identifying matching
          patterns.
        2 Select the pair of matching patterns which represents the greatest
          amount of redundancy.
        3 Unify the selected pair of matching patterns.
END
```

The iteration of these three operations constitutes a simple form of hill-climbing search.

4. Examples

The examples described in this section are intended to illustrate how SP6 may perform both inductive learning and logic. We stress that **these two contrasted kinds of computing result from one unifying concept: a search for efficiency in knowledge.**

As already mentioned, bold type is used in the examples to highlight symbols which have been created by unification of two or more other symbols. And the symbol '=>' used in the examples is not part of the SP language but is used to show how one SP object is transformed into another in the course of computing.

4.1. Inductive Learning

We are using the term *inductive learning* to mean a process of organising knowledge which is done without explicit instruction or intervention from a (human or non-human) 'teacher' and without any other knowledge apart from the learning processes themselves.

4.1.1. Example 1: Discovering Word Segments, Classes and Shared Structure

```
[
        (p e t e r m e e t s j a n e)
        (p e t e r a d m i r e s j a n e)
        (p e t e r m e e t s p e t e r)
        (p e t e r a d m i r e s p e t e r)
        (j a n e m e e t s p e t e r)
        (j a n e a d m i r e s p e t e r)
        (j a n e m e e t s j a n e)
        (j a n e a d m i r e s j a n e)
]
=>
([%1 {(p e t e r)(j a n e)}] {(m e e t)(a d m i r e)} s [%1 _])
```

The first example illustrates how, from examples of a simple sub-set of English, the SP system infers a structure which is equivalent to a grammar for the sub-set.

The OAOs in the first SP object may be seen as ‘sentences’. By analogy with the way people normally talk, there is no explicit marker for the end of one word within a sentence and the beginning of the next. Likewise, there is nothing explicit in the example sentences to mark classes of syntactically equivalent words.

SP6 transforms the first object into the second by searching for redundancy and removing it wherever possible. The effect of this search is to identify the words within the sentences as discrete entities and to group the words into syntactic classes. The search process also has the effect of isolating the terminal ‘s’ in ‘meets’ and ‘admires’ rather in the same way that a linguist would identify that terminal ‘s’ as a distinct sub-structure (‘morpheme’) within these words.

SP6 recognises that the ORO {(p e t e r)(j a n e)} occurs in two contexts, at the start of each sentence and at the end. It removes the redundancy by placing the structure arbitrarily in one of the two locations (first in the example) giving it a label (‘%1’) and leaving this label as a ‘reference’ at the other location (last) where the structure can occur. A tag like ‘%1’ is equivalent to a label like ‘NP’ used in conventional grammars.

4.1.2. Example 2: Induction of a Class Hierarchy

```
[
        (backbone cold-blooded scaly lungs dry-skin limbless
                wide-gape snake)
        (backbone warm-blooded fur milk whiskers purrs retractile-claws cat)
        (backbone cold-blooded scaly lungs dry-skin limbs lizard)
        (backbone warm-blooded feathers beak wings flightless ostrich)
        (backbone cold-blooded scaly swims gills fresh-water small minnow)
        (backbone warm-blooded fur milk barks dog)
        (backbone warm-blooded feathers beak wings flies
                small brown sparrow)
        (backbone cold-blooded scaly swims gills sea-living aggressive shark)
]
=>
(backbone
        {(cold-blooded scaly
                {(lungs dry-skin
                        {(limbless wide-gape snake)
                        (limbs lizard)})
                (swims gills
                        {(fresh-water small minnow)
                        (sea-living aggressive shark)})})
        (warm-blooded
                {(feathers beak wings
                        {(flightless ostrich)
                        (flies small brown sparrow)})
                (fur milk
                        {(whiskers purrs retractile-claws cat)
                        (barks dog)})})})
```

This example illustrates how SP6 may create a class hierarchy or taxonomic hierarchy in the manner of numerical taxonomy or cluster analysis. The example also illustrates how SP6 can create a 'discrimination net' like the ones produced by Quinlan's ID3 (Quinlan, 1983).

The first object shows a set of animals, each with some distinctive attributes. OAOs have been used rather than the more appropriate UAOs because SP6 lacks the ability to look for patterns and sub-groupings within UAOs. Future versions of SP will plug this gap.

The second object is, in effect, a class hierarchy or discrimination net with attributes at appropriate levels. In the spirit of object-oriented design (Cook, 1986), redundancy is minimised by placing each attribute at a level in the hierarchy where it applies to all the lower levels and none of the higher levels.

4.2. Logic and Logical Inferences

The two examples in this section illustrate how the SP language may be used to express logical propositions and how, as a by-product of its search for efficiency in information, the SP system may do the kinds of things which are normally regarded as logical inference.

In this area, SP6 has a weakness which we believe we can cure in future versions. SP6 is designed so that variables (shown with the symbol '_') will match and unify with one SP object and only one object. We now believe that the number of objects to be matched and unified with any given variable should not be set at an arbitrary value but should be determined by the search for efficiency. This would mean that any given variable would match and unify with zero or more SP objects.

4.2.1. Example 3: a Simple Syllogism

[[mortal [man _]] [man Socrates]]
=>
[mortal [**man** Socrates]]

This example is modelled on the well-known syllogism: "All men are mortal. Socrates is a man. Therefore, Socrates is mortal." In logical notation the syllogism is written like this:

$$\forall x \in \text{man} . \text{mortal}(x) \wedge \text{Socrates} \in \text{man} \Rightarrow \text{mortal(Socrates)}$$

Given the current meaning of '_' as 'one and only one', the object [man _] may be unified with any UAO which contains the symbol 'man' and one other object. Thus it may be read as "any man with one attribute" rather than the more general "any man" or "all men".

If we ignore the restriction on the meaning of [man _] then [mortal[man _]] should be read as "Any man is mortal" or "All men are mortal".

[man Socrates] should be read as "Socrates is a man".

The object created by SP, [mortal[man Socrates]], should be read as "Socrates is a man and he is mortal". This covers the meaning of the conventional conclusion ("Socrates is mortal") and also includes one of the premises of the syllogism - that "Socrates is a man". There are two things here which deserve comment:

- The first point is that any version of SP which is designed to preserve power in a knowledge structure - like SP6 - should preserve logical premises along with any inferences which may be drawn. The apparent difference between this style of working and the conventional style (where only the inference is shown) is more cosmetic than real. Conventional systems for logic do normally preserve logical premises but only display the inferences. The display mechanisms in SP could, perhaps, be designed to pick out inferences for attention in a similar way.
- The second point is that this version of SP6 has actually failed to preserve one of the premises in this example (that "All men are mortal") although, in designing the system, we intended that it should not lose power in its knowledge structures. We expect this weakness in SP6 to be cured in later versions of SP when the meaning of '_' is changed to become 'zero or more objects'. In this case, the SP system should work like this:

[[mortal[man _]] [man Socrates]]
=>
[mortal[**man** Socrates _]]

The variable remains in the final object because unification with ‘Socrates’ does not exhaust the ‘or more’ part of its meaning. The result should be read as "Socrates, and all other men, are mortal", a conclusion which covers the meaning of "Socrates is mortal" and preserves the meaning of both of the original premises.

4.2.2. Example 4: Transitive Relations

[[equals A B _ _] [equals B C _ _] [equals C D _ _]]
=>
[**equals C** D A **B**]

This example illustrates how, from the premises that "A equals B and B equals C and C equals D", the SP system can infer that A, B, C and D are all equal.

Like the previous example, this example is not completely accurate because the variable (‘_’) means ‘one and only one’ instead of ‘zero or more’. Given the current meaning of the SP variable, an object like [equals A B _ _] means "A and B and two other unspecified objects are all equal". The two instances of ‘_’ are needed in this example for unification with C and D.

If ‘_’ had its proper meaning then a transitive relation like "A equals B" would be represented in SP as [equals A B _] which may be read as "A equals B and there may be other objects which are equal to A and B". The variable shows that unification may add other objects to the relationship and thus marks the relationship as transitive. By contrast, an intransitive relationship such as ‘likes’ may be represented as (likes A B) - the absence of a variable in this structure is what ensures that the relationship is not transitive. (This relationship is shown as an OAO because the fact of A liking B does not mean that B likes A.)

Notice that this example would not give a valid conclusion if the relationship were, say, ‘not-equals’. This and other examples point to the conclusion that negation will require special treatment in SP, a point discussed below.

4.2.3. A Comparison with Prolog

To represent a transitive relationship like ‘equals’ in an established logic programming system like Prolog, it is necessary to include a rule like this:

```
equals(X,Z) :- equals(X,Y), equals(Y,Z).
```

This rule means "X equals Z if X equals Y and Y equals Z"; X, Y and Z are all variables. The inclusion of this rule along with ‘facts’ like equals(a,b) and equals(b,c) allows the system to infer that equals(a,c) is true.

This way of representing the transitive nature of ‘equals’ works well enough but is distinctly more cumbersome than what is needed in SP. In addition, a rule like the one just shown needs to generalized to cope with other transitive relations like ‘ancestor’, ‘before’, ‘above’, ‘bigger’ etc. and the generalization is relatively complicated.

SP seems to have a distinct advantage over established systems in this area.

4.2.4. TRUE and FALSE in SP

At present, SP has no explicit concepts of TRUE and FALSE. As with systems like Prolog, we may adopt the ‘closed world hypothesis’ and suppose that truth is whatever the system knows about and everything else is false. In this case, the answer to a 'query' is 'true' if the query unifies with an object in a given knowledge structure, otherwise it is 'false'.

In SP it is natural to generalise this model of truth to take advantage of the fact that SP can recognise partial matches between structures as well as full matches. Given this feature of SP, we may recognise degrees of truth and falseness. If a query unifies completely with an object in the knowledge base then it is unequivocally true in terms of that knowledge base. If it does not unify with anything in the knowledge base then it is unequivocally false. In all other cases, it has some degree of truth corresponding to the amount of unification which has been achieved.

Does SP need a negation operator ('not') and, if so, what is its status within the SP theory? In the spirit of 'simplicity and power' it would be nice if negation in SP could be a by-product of mechanisms which are already established in the system.

One possibility is that 'not' is simply a symbol (with the same status as all other symbols) which may be used as a 'spoiler' to prevent a given object unifying with any identical copy of that object. Thus [not TRUE] would not unify with TRUE and would therefore be 'false' in the terms which we have sketched.

One difficulty with this idea is that the 'spoiler' would never eliminate partial unification and would thus never have the meaning 'unequivocally false' as one would wish. Another problem is that [not [not TRUE]] would not mean the same as TRUE as one would wish.

These points and examples like 'not-equals', mentioned above, suggest that a special negation operator will be needed in SP. This and related questions are issues to be explored in the future.

5. Conclusion

In this paper we have tried to show how two sub-fields of computing which are not normally related may be seen as two facets of a more general set of principles. All the examples which have been shown - examples of inductive learning and examples of logical inference - are the product of a system which is dedicated to one goal: searching for efficiency in information by finding redundancy in the information and removing it wherever possible.

There are many issues still to be explored in developing this framework of ideas but there are also many potential benefits - in simplifying our thinking about computing and in the practicalities of using computers. The practical benefits and advantages which expect from the SP system when it is mature are described in Wolff & Chipperfield (submitted for publication).

Acknowledgement

We are very grateful to Paul Mather for helpful comments on earlier drafts of this paper.

References

Baeten J C M, Bergstra J A & Klop J W (1987). Term re-writing systems with priorities. Proceedings of the Conference on Re-writing Techniques and Applications, Bordeaux, France, May 1987. pp 83-94.

Campbell C, Sherrington D & Wong K Y M (1989). Statistical mechanics and neural networks. In I Aleksander (Ed), *Neural Computing Architectures*, London: North Oxford Academic.

Chaitin G J (1987). *Algorithmic Information Theory*. Cambridge: Cambridge University Press.

Clocksin W F & Mellish C S (1981). *Programming in Prolog*. Heidelberg: Springer-Verlag.

Cook S (1986). Languages and object-oriented programming. *Software Engineering Journal 1 (2)*, 73-80.

Hinton G E & Sejnowski T J (1986). Learning and relearning in Boltzmann machines. Chapter 7 in D E Rumelhart and J L McClelland (Eds), *Parallel Distributed Processing*, Vol I, Cambridge Mass.: MIT Press, pp 282-317.

Lavington S H (1988). An overview of knowledge manipulation engines and the Intelligent File Store. Proceedings of the IEE Colloquium on Knowledge Manipulation Engines, London. pp 1-4.

McGregor D, McInnes S and Henning M (1987). An architecture for associative processing of large knowledge bases (LKBs). *The Computer Journal 30(5)*, 404-412.

Quinlan J R (1983). Learning efficient classification procedures and their application to chess end games. In R Michalski, J G Carbonell and T Mitchell (Eds) *Machine Learning: an Artificial Intelligence Approach*. Palo Alto, CA: Tioga.

Risch T, Reboh R, Hart P & Duda R (1988). A functional approach to integrating database and expert systems. *Communications of the ACM 31*, 1424-1437.

Robinson J A (1988). Beyond Loglisp: combining functional and relational programming in a reduction setting. *Machine Intelligence 11*, 57-68.

Shieber S M (1986). *An Introduction to Unification-Based Approaches to Grammar*. Stanford, CA: Center for the Study of Language and Information.

Siekmann J H (1989). Unification theory. *Journal of Symbolic Computation 7*, 207-274.

Wolff J G (1987). Cognitive development as optimisation. In L Bolc (Ed), *Computational Models of Learning*. Heidelberg: Springer-Verlag, 161-205.

Wolff J G (1988). Learning syntax and meanings through optimisation and distributional analysis. In Y Levy, I M Schlessinger and M D S Braine

Wolff J G (1989). Information and redundancy in computing and cognition.*AISB Quarterly 68*, 14-17.

Wolff J G (in press). Simplicity and power: some unifying ideas in computing. To appear in the *Computer Journal*.

Wolff J G & Chipperfield A J (in preparation). SP: a new view of software engineering and computing.