GSG No. 52
Erik Sandewall
December, 1968

USE OF AMBIGUITY LOGIC IN THE PICTURE DESCRIPTION LANGUAGE

Abstract: The Picture Description Language is extended by the use of the ambiguity operator and the ro-notation (a variant of the lambda-notation). This enables us (1) to avoid the labeling scheme, which was an ad hoc notation; (2) to express attributes in predicate calculus notation; and (3) to write recursive descriptions of pictures (such as a page of text) inside the language.

1. Introduction

We consider the following characteristic facts about Alan C. Shaw's Picture Description Language /3,5/:

(1) Each sentence in the language is an expression, built from primitive names and concatenation operators (+, x, -, *).

(2) Each primitive name stands for any member of a class of pictures. The same is true about each expression.

(3) If S1 and S2 are expressions, and Θ is an operator, then S1 Θ S2 stands for a class of pictures which are formed from each possible combination of a picture from the class S1 and a picture from the class S2. (The * operator is an exception).

(4) If the same primitive name occurs several times in the same expression, there is no restraint against selecting different members of the class in those two instances.

(5) In many cases we want to select one arbitrary member of a class of pictures, and use that same member several times in the same expression. We call this the multiple use problem. It is handled, in Shaw's original PDL/5/, by a generalized concatenation operator, and in the later version /3/, by a labeling scheme.

In our opinion, both these notations for handling the multiple use problem are formally "impure", and it is desirable to find a more systematic notation. It turns out that the ambiguity logic, which was investigated by the present author for quite different purposes /4/, can be useful for the multiple use problem.

The following chapter contains an exposition of the ambiguity logic. In chapter 3, its application to picture description is discussed.

2.1 The basic notation

In /2/, John McCarthy briefly proposes the use of an ambiguity operator. In this chapter, we shall develop a logic based on such an operator.

Let us use the question-mark (?) as an infixed, binary operator, and let "a ? b" mean "the ambiguous expression whose value is a, or b". We have to

make this more precise. When we write "f(x) = a?b", we could mean:

(1) The value of f(x) may be a, and it may be b. We don't know (or we don't care to tell).

(2) f(x) has two values. One of them is a; the other is b.

Some axioms hold in either case, viz.

(A1) a ? a = a

(A2) a ? b = b ? a

(A3) a ? (b ? c) = (a ? b) ? c

(A4) f(a ? b) = f(a) ? f(b)

The difference between (1) and (2) becomes acute when we are asked whether we want to assume an axiom like

(A?) $a = b \supset a = (b\ ?c)$

which would permit us to write e.g.

$$2 + 3 = 4\ ?\ 5.$$

It is more fruitful to consider this as a question: What do we intend to mean by equality (between ambiguous expressions)? If accepted, (A?) would violate the transitivity of equality, because we would have

$$3 = 3\ ?\ 4 = 4 = 4\ ?\ 5 = 5 \quad \text{etc.}$$

Therefore, we reject "axiom" (A?) and choose interpretation (2) above. This also means that we would <u>not</u> write

$$\forall\ ((n),\ \mathrm{ult}(n) = 0) = T$$

as McCarthy does in /2/. (We would instead write $\forall\ ((n),\ 0 \leftarrow\!\!\cap\ \mathrm{ult}(n))$; see below).

Axiom (A3) makes it natural to omit parenthesis in sequences of ?'s and write e.g. "a ? b ? c ? d". We shall say that this is an <u>ambiguous expression</u>. It <u>denotes</u> an ambiguous object or <u>ambject</u> with the <u>cases</u> a,b,c,d, a?b, a?c, ... a?b?c?d. If we want a strict definition of equality, we can assume a "weak" equality between unambiguous expressions to be given, and define a strong equality as follows: a = b if every unambiguous case of a equals some unambiguous case of b, and vice versa.

We shall use the symbol $\mathcal{R}$ for the "inconceivable case", so that

$a \,?\, \mathcal{R} = a$

If e is an expression, let us write $*e$ [1] for "e contains only itself and $\mathcal{R}$ as cases" or "e is unambiguous". We also stipulate $\neg * \mathcal{R}$, and then obtain as an axiom

(A5) $*(a \,?\, b) \supset a = b$

We shall distinguish between regular and dominant functions. A regular function is a function which satisfies axiom (A4) above; a dominant function is one which does not. In particular, ? itself is a regular function. Functions of several arguments can be regular in some of them.

The same distinction is valid for predicates. A regular predicate is one which satisfies

$p(a \,?\, b) = p(a) \,?\, p(b)$.

If $p(a) =$ true, $p(b) =$ false, we have $p(a?b) =$ true ? false. Here, ? is a logical connective, rather than a function. We obtain a four-valued logic with the values true (t), false (f), t ? f or sometimes (s), and $\mathcal{R}$, $\wedge$, $\vee$, $\neg$, etc. are regular connectives in this logic, and we have e.g.

$\underline{t} \,?\, \underline{f} = \underline{s}$

$\underline{s} \,?\, \underline{t} = (\underline{t} \,?\, \underline{f}) \,?\, \underline{t} = \ldots = \underline{s}$

$\underline{s} \wedge \underline{t} = (\underline{t} \,?\, \underline{f}) \wedge \underline{t} = (\underline{t} \wedge \underline{t}) \,?\, (\underline{f} \wedge \underline{t}) = \ldots = \underline{s}$

It is a trivial task to write down the truth-tables for these four truth-values and the logical connectives $\wedge$, $\vee$, $\neg$, ? . See appendix. We may also use a dominant or strong implication, defined in analogy with strong equality; u implies v strongly (written $u \supset\!\!\!\supset v$) if every unambiguous case of u implies weakly (i.e. with a regular function) some unambiguous case of v, and every unambiguous case of v is implied by some unambiguous case of u. It follows that strong implication is transitive; that modus ponens still holds; and that

$f \supset\!\!\!\supset s \supset\!\!\!\supset t$.

The truth-table of strong implication is also given in the appendix.

(1) This one-argument * should be distinguished from the two-argument * of the PDL.

A purist might prefer to use different symbols for the function ? and the logical connective ?. However, axioms (A1) - (A5) are valid for both these kinds of ? (if the function symbol f is axiom (A4) is interpreted so that wffs are obtained). We therefore use the same symbol and save duplication of axioms.

By the same philosophy, we consider = and * as dominant predicates and, at the same time, as dominant logical connectives. We have e.g.

$(\underline{t} = \underline{s}) = \underline{f}$

$*(u = v)$

We need one more function, a dominant function / which is similar to intersection between sets in the sense that a / b shall be an ambiguous expression whose cases are the common cases of a and b. This gives us e.g.

$(a\ ?\ b)\ /\ a\ = a$

$f(a\ /\ b)\ = f(a)\ /\ f(b)$ if f is regular.

If a and b have no common cases, a/b is $\mathcal{R}$. Likewise, $f(\mathcal{R}) = \mathcal{R}$ for regular functions f.

Finally, we need one dominant predicate, corresponding to set-inclusion. $a \hookleftarrow b$ shall be $\underline{t}$ if every case of a is also a case of b (the "vice versa" is not required) and $\underline{f}$ otherwise. This gives us

$a\ /\ b \hookleftarrow a \hookleftarrow a\ ?\ b$

$(a \hookleftarrow b) \wedge (b \hookleftarrow a) \supset (a = b)$

Of course, / and $\hookleftarrow$ have a second nature as logical connectives.

The ambiguity concept plays us a few tricks. We no longer have

$(u \vee \neg u)$

as an axiom. More significant is the impact on the λ -notation, which is the topic of the next section.

2.2 The λ-notation and the ρ-notation

Should we consider a λ-expression as a regular or a dominant function? Let us see what happens when we apply axiom (A4):

$p(a,a)\ ?\ p(b,b) \quad =$
$[\ \lambda\ (t)p(t,t)]\ (a) \quad ?\ [\ \lambda\ (t)p(t,t)]\ (b) \quad =$ /axiom (A4) !/
$[\ \lambda\ (t)p(t,t)]\ (a\ ?\ b) \quad =$
$p(\ a?b,\ a?b) \quad =$
$p(a,a)\ ?\ p(a,b)\ ?\ p(b,a)\ ?\ p(b,b).$

Thus it seems that axiom A4 is not applicable, and that λ-expressions should be considered as dominant functions. However, we may instead question the third equality sign in the example, which relies on the axiom

(K) $[\ \lambda\ (t)\ F(t)]\ (a) \quad = F(a)$

If we take as a requirement in axiom K that a must be unambiguous, then lambda-expressions will indeed be regular functions. We shall permit both cases, and introduce the symbol ρ for the lambda that creates regular functions. Thus every expression $(\ \rho\ (\ldots)\ \ldots)$ is a regular function, and we have the axiom

$$*a \supset ([\ \rho\ (t)\ \ F(t)]\ \ (a) \quad = F(a)\ )$$

λ-expressions, on the other hand, are dominant functions and satisfy axiom K.

This trouble with λ-expressions may seem a nuisance. In fact, it is an asset, because we can use ρ in many cases where a universal quantifier would otherwise be used. Let us work a simple example in ordinary logic notation.

Ex. Let B1, B2, ... Bn be boys, and define the function
father(x) for "the father of x"
and the predicate
admire(x,y) for "x admires y".

The phrase "all boys admire their fathers" is usually written

$$\forall ((b) \quad b \in Bin \supset admire(b, father(b)) \quad)$$

where $Bin = \{ B1, B2, \ldots Bn \}$.

With the ρ -notation, we can instead write

$$[\rho (b) \quad admire(b, father(b))] (boy)$$

where boy = B. ? B2 ? ... Bn.

This works as follows. Let A be the regular predicate

$$[\rho (b) \; admire (b, father(b))]$$

Our axiom states that A(B1 ? B2 ? ... ? Bn) is t, but as the argument is ambiguous, we cannot substitute it into the definition of A. But according to regularity, we have instead

$A(B2) \leftharpoonup A(B1 ? B2 ? \ldots ? Bn) = t$, and through application of a sequence of obvious axioms, we can conclude

$$A(B2) = t$$

As we also have *B2, we can now use the axiom for ρ and conclude

admire(B2, father (B2))

In an actual use of this notation, we would of course not feel compelled to define "boy" through an enumeration of all possible boys. Instead, we would assume the ambiguous object "boy" and use the function A each time we encountered an object Bk with the properties $Bk \leftharpoonup boy$, *Bk.

2.3 The let-notation

The lambda-notation (and by consequence, the ro-notation) often yields expressions that are difficult to read, especially when the lambdas are nested. The let-notation, originally proposed by Landin /1/, is an equivalent notation with increased legibility.

A few examples will make the notation clear.

Ex. 1 lambda-notation: $[\lambda (x) \quad f(x,x)] (a)$

let-notation: let x be a in f(x,x)

Ex. 2 lambda-notation: $[\lambda (x,y) \; f(x,y) + f(y,x)] (a,b)$

let-notation: let x be a, y be b in f(x,y) + f(y,x)

Ex. 3 lambda-notation: $[\lambda (x), [\lambda (y,z) \; f(y,z) + f(z,y)] \quad (f(x,x), g(x))](g(a)$

let-notation: let x be g(a) in

let y be f(x,x), z be g(x) in f(y,z) + f(z,y)

We shall extend the notation to ro-expressions, but distinguish them by writing be a or (synonymously) be an instead of be.

Ex. 4 ro-notation: the example above
let-notation: let b be a boy in admire (b, father(b))

2.4 Ambiguous Objects and Sets

As we saw in the example in section 1.2, the symbols /, ?, ↩ , etc. can be used for forming and describing collections of objects. Usually, we use set notation for such purposes. If the reader sees the difference between the two notations, please proceed to the next chapter; else read the following very informal characterization.

In the ambiguity notation, if we form a collection of unambiguous objects, A1 ? A2 ? A3, and another collection, B1 ? B2, and then form a collection from these, we obtain (A1 ? A2 ? A3) ? (B1 ? B2), where the parentheses can be removed. In set notation, we obtain instead $\{\{A1, A2, A3\}, \{B1, B2\}\}$, where the subsets keep their individuality.

Figuratively speaking, forming a set of a number of objects means putting them into a box (which can then, as a unit, be put into other boxes), whereas forming an ambject means making a heap of the objects. If two heaps are put together, they lose their identity.

With this analogy, Я is the empty heap, and *A for a heap a means "A consists of only one element". Dominant functions and predicates are those that operate on heaps as wholes; regular functions and predicates are those that operate individually on each member of a heap.

Ambiguous objects could be expressed entirely in terms of sets, with the ? operator expressed as ⌣ , with * being the predicate "set has only one member", etc. however, the use of regular predicates would then force us to introduce such novelties as sets of truth-values. It appears more convenient to distinguish between sets and ambjects.

3.1 Ro-notation and the Multiple Use Problem

We shall demonstrate that the ro-notation takes care of the multiple use problem. Before we turn to the Picture Description Language, let us show how the ambiguity operator and the ro-notation works on arithmetic expressions (For convenience, we use the equivalent let-notation) :

1. $(3 \,?\, 4) + 5 \;=\; 3+5 \,?\, 4+5$
2. $(3 \,?\, 4) + (5 \,?\, 6) \;=\; 3+5 \,?\, 3+6 \,?\, 4+5 \,?\, 4+6$
3. let t be a $3 \,?\, 4 \,?\, 5$ in $t + 2 \;=\; 3+2 \,?\, 4+2 \,?\, 5+2$
4. let t be a $3 \,?\, 4$, u be a $5 \,?\, 6$ in $t + u \;=\; (3 \,?\, 4) + (5 \,?\, 6)$
5. $(3 \,?\, 4) + (3 \,?\, 4) \;=\; 3+3 \,?\, 3+4 \,?\, 4+3 \,?\, 4+4$
6. let t be a $3 \,?\, 4$ in $t + t \;=\; 3+3 \,?\, 4+4$
7. let t be a $3 \,?\, 4 \,?\, 5$, u be a $2 \,?\, 7$ in $t \times (u + 2) \;=$
   $3 \times (2 + 2) \;?\; 4 \times (2 + 2) \;?\; 5 \times (2 + 2) \;?$
   $3 \times (7 + 2) \;?\; 4 \times (7 + 2) \;?\; 5 \times (7 + 2)$

8. Assume "odd" is the name for the infinite expression $1 \,?\, 3 \,?\, 5 \,?\, 7 \,?\, \ldots$
   Then
   let t be an odd in $t \times (t-1) \;=\; 1\times0 \;?\; 3\times2 \;?\; 5\times4 \;?\; 7\times6 \;?\; \ldots$

In all these examples we have ambiguous expressions, which stand for collections of integers (e.g. $3 \,?\, 4 \,?\, 5$), and through the use of the ro-notation, we are able to pick out an arbitrary member of that collection, use it several times in the same expression, and form a new collection by applying this expression to each member of the original collection. This is exactly our multiple use problem for the Picture Description Language.

The reader should make sure he understands how this works. Let us perform all intermediate steps in example 6:

let t be a $3 \,?\, 4$ in $t + t$ = /defn of let-notation/
$[\rho\ (t)\ t+t]\ (3 \,?\, 4)$ = / ρ gives regular functions/
$[\rho\ (t)\ t+t]\ (3) \;?\; [\rho\ (t)\ t+t]\ (4)$ = /integers are unambiguous, i.e. *3 and *4/

$3+3 \;?\; 4+4$

To apply this to the PDL, we make the following conventions:

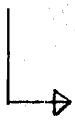
(1) Primitive names are interpreted as names of ambiguous objects, possibly with infinitely many cases: the possible pictures.

(2) The concatenation operators (+, x, -, *) are regular functions in both their arguments.

It immediately follows that expressions also denote ambjects. For example, if "seg" is the name for one class of picture elements, and "diag" the name for another such class, "seg + diag" denotes an ambject whose cases are obtained by adding (with the + operator) an arbitrary member of the class seg with an arbitrary member of the class diag.

For another example , let us introduce the following picture elements:

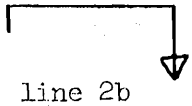
line 1a



line 1b

line 1c

line 2a



line 2b

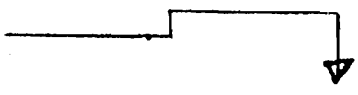
Each name stands for one unique picture element, and we therefore have *line 1a, *line 1b, ... *line 2b. Let us define

line 1 = line 1a ? line 1b ? line 1c

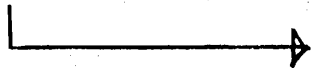
line 2 = line 2a ? line 2b

Line 1 and line 2 are both primitive classes (Shaw's terminology) or ambiguous objects (our terminology).

The operator + is a regular function. On un-ambiguous objects (pictures), we can immediately apply the definition:



line 1a + line 2b

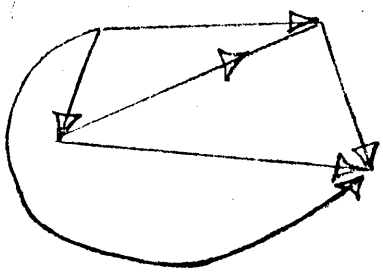


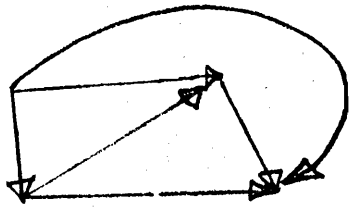
line 1c + line 1a

On ambiguous objects, we use regularity. Thus line 1 + line 2 is a primitive class/ambiguous object with six cases, obtained by adding any case of line 1 to any case of line 2. Other operators operate in the same manner (except *, which is treated in the next section).

Line 1 and line 2 are primitives with a finite number of cases, but there is nothing to prevent us from introducing names for primitives which contain an infinite number of cases. Such primitives can be described with predicates, but cannot be defined through enumeration.

Sometimes, it is necessary for the description to refer several times to the same component of a graph. As an example, we shall write an expression for "the" complete four node graph, i.e. the ambject which has as cases various four node graphs like





(As there are many different complete four node graphs, it is very reasonable to say that "the complete four node graph" is an ambiguous description). Let linseg be "the" straight line (i.e. the ambject which has all possible straight lines as cases), and let arc be "the" arc. We assume that five out of the six arrows are straight lines, and that the last one is an arc. Without the straight line diagonal, the picture can be described as

(linseg + linseg) * (linseg + linseg) * arc;

insignificant parentheses have been omitted. With both diagonals, we introduce the name "upper" for the upper, horizontal line segment, and "diag" for the diagonal, and obtain the expression

let upper be a linseg, diag be a linseg in
(upper + linseg) * (((linseg + diag) - ⌈upper⌉) + (~ ⌈diag⌉) + linseg) * arc

Reading the expression from left to right, we see that the following happens: First we follow the  part of the picture; then we go back to the upper left-hand corner; go down and up the diagonal to the head of "upper" (remember that a-b has the same tail and head as a, but the head must also be the head of b); back the same diagonal, but now with all points eliminated, and along to the lower righthand corner; finally we go along the arc.

It is easily seen that any graph can be described by walking around this way.

3.2 The Constant Я and the * operator

The picture corresponding to an expression a * b can not always be constructed. Shaw, in his first memo /5/, handles this by assuming that "the * operator implies a semantic restraint between its two operands" (page 15).

With the ambiguity logic, we can express this more formally. If a and b are unambiguous, i.e. refer to unique pictures, and if a*b cannot be constructed, we make the convention that a*b = Я . In the example of the previous section, we have e.g.

line 1b * line 2a = Я .

The reason for this convention becomes obvious when some argument of * is not unambiguous. Remember that Я was introduced with the rule a ? Я = a. Using again the example of the previous section, we therefore have e.g.

line 1 * line 2 = ... = line 1a * line 2a ? Я ? Я ? Я ? Я ?
? line 1c * line 2b =
= line 1a * line 2a ? line 1c * line 2b

In general, a*b is an ambiguous object which contains as cases all geometrically possible combinations (with the * operator) of cases of a and b. This is what we would intuitively desire.

3.3 Using Predicates Instead of Attributes.

Primitives are names for classes of pictures, and it is therefore necessary to define somehow the extension of this class. Sometimes it may be sufficient to define the class implicitly by the recognition function that recognizes members of the class; sometimes we desire a more explicit specification.

Shaw suggest that this be made through the use of attributes. Each primitive class is defined through a number of attributes, which may indicate form, size, direction, etc.

This approach has the disadvantage that there does not exist any calculus for manipulating such attributes, and we are therefore left without help if we want (for instance) to speak about the attributes of a+b, as inferred from the attributes of a and b, individually.

With the ambiguity logic, we can solve this problem by using predicates instead of attributes. Let us return for a moment to the examples with integers. The predicate Odd(n) is defined to be true if the integer n is odd, and false if n is even. For example, Odd(3) = true. If we now declare Odd to be a regular predicate, we have e.g. Odd(3 ? 5) = Odd(3) ? Odd(5) = true ? true = true. Similarly, using the constant "odd" (+ 1 ? 3 ? 5 ? ...) from section 3.1, we have Odd(odd) = (an infinite sequence of true's) = true.

Conversely, if we have as axioms Odd(odd) and 1 $\leftarrow\!\!\cap$ odd, 3 $\leftarrow\!\!\cap$ odd, etc. (and also $^{\kappa}t \supset {}^{\kappa}\mathrm{Odd}(t)$), we can infer Odd(1), Odd(3), etc.

When we want an alternative to the use of attributes in picture calculus, we use the latter direction of inference. For instance, suppose that we have a regular predicate straightline of one argument, and that straightline(t) for unambiguous t is true if t is a straight line. For example, we have straightline(line 1a), above. To express the fact that every case of the ambiguous object linseg is a straight line, we write straightline(linseg). For each case t of linseg, we can conclude straightline(t) in analogy with the inference for Odd, above.

Similarly, if length(s) is a regular function that evaluates to the length of a line segment along its path, and $\leq$ is taken as a regular predicate in both its two arguments, then the restriction "all linsegs are at least two length units long" is expressed through $2 \leq \text{length(linseg)}$.

This is of course only a theoretical remark. In an actual computer representation, it is probably desirable to store information about primitives on a LISP-type property-list, i.e. as attributes.

3.4 Recursive Definitions of Classes of Pictures

In the memos /3/ and /5/, three types of pictures are handled completely differently:

(1) Primitive classes. They are given a name and are described by a set of attributes.

(2) Non-recursive classes. These are classes which can be formed from the primitive classes through a finite number of application of the operations +, -, x, etc. Each class is described by an expression in the PDL. The properties of the members of a class can be deduced from this expression, and from the attributes of the primitives in the expression, but this has to be done outside the PDL.

(3) Recursive classes. An example is the class of pages of printed text. It is impossible (or at least impracticable) to write a closed expression for these classes. Instead, Shaw suggests (in /5/) that we should write a grammar which generates an infinite set of closed expressions, and which has the property that the set of all cases of all generated expressions is equal to the given recursive class.

With the introduction of predicates instead of attributes, it became possible to apply predicates to expressions (case 2), so that much of the distinction between (1) and (2) vanished. In this section, we shall demonstrate that recursive classes can be described with the use of the ambiguity operator, and completely inside the language (i.e. without resorting to a grammar).

We use the same example and the same elements as Shaw in /5/, fig. 6, and write down the definition immediately:

page = start + lines + end
start = $e_v + e_m$
end = $e_v + \perp$
lines = line ? line x (e_l + lines)
line = words - eol
words = word ? word + e_w + words
word = char ? char + e_c + word
char = a ? b ? c ? ... ? y ? z
eol = ⊣ + e_h

All three recursions clearly work the same way. Let us look at the last one, defining words from characters. First, by the axiom

a ↶ a ? b

we conclude that

char ↶ word

i.e. all characters are words. Second, by the same axiom, the newly-won insight, and the regularity of +, we have

char + e_c + char ↶ char + e_c + word ↶ word

i.e. any sequence of two characters is a word. That conclusion enables us to say

char + e_c + char + e_c + char ↶ char + e_c + word ↶ word

and so on.

As we see, groups (1) and (3) above are now handled in essentially the same way. Primitive classes (1) have names, and are described by the use of functions and relations, e.g. "2 ≤ length (linseg)". Recursive classes (3) also have names, and are described by functions and relations, e.g. "line = words - eol". The difference, of course, is that the relations that describe recursive classes are less explicit.

In conclusion, use of the ambiguity logic enables us to reduce various notational devices in the Picture Description Language to a coherent system of predicate-calculus-like notation. This could simplify the set of axioms for the PDL and facilitate making a program for manipulating the PDL.

References

1. P.J. Landin
A Correspondence Between Algol 60 and Church's Lambda notation,
Part I Comm. ACM 8 (1965), p. 89

2. John McCarthy
A Basis for a Mathematical Theory of Computation in
P. Braffort and D. Hirschberg (eds)
Computer Programming and Formal Systems, Amsterdam, 1963.

3. W.F. Miller and Alan C. Shaw
A Picture Calculus
GSG No. 40

4. Erik J. Sandewall
Ambiguity Logic as a Basis for an Incremental Computer
Forthcoming Stanford A.I. Project Memo

5. Alan C. Shaw
A Proposed Language for the Formal Description of Pictures
GSG No. 28

Appendix 1: Truth-values for logical connectives in the ambiguity logic

Truth-values: t = true, f = false, s = sometimes.

a	b	a∧b	a∨b	a⊃b	a?b	a/b	a=b	a⊃b	a↩b
t	t	t	t	t	t	t	t	t	t
t	s	s	t	s	s	t	f	f	t
t	f	f	t	f	s	ℛ	f	f	f
t	ℛ	ℛ	ℛ	ℛ	t	ℛ	f	f	f
s	t	s	t	t	s	t	f	t	f
s	s	s	s	s	s	s	t	t	t
s	f	f	s	s	s	f	f	f	f
s	ℛ	ℛ	ℛ	ℛ	s	ℛ	f	f	f
f	t	f	t	t	s	ℛ	f	t	f
f	s	f	s	t	s	f	f	t	t
f	f	f	f	t	f	f	t	t	t
f	ℛ	ℛ	ℛ	ℛ	f	ℛ	f	f	f
ℛ	t	ℛ	ℛ	ℛ	t	ℛ	f	f	t
ℛ	s	ℛ	ℛ	ℛ	s	ℛ	f	f	t
ℛ	f	ℛ	ℛ	ℛ	f	ℛ	f	f	t
ℛ	ℛ	ℛ	ℛ	ℛ	ℛ	ℛ	t	t	t

a	¬a	*a
t	f	t
s	s	f
f	t	t
ℛ	ℛ	f