OPERATIONAL POINTER SEMANTICS: SOLUTION TO SELF-REFERENTIAL PUZZLES I

Haim Gaifman
Mathematics and Computer Science Institute
Hebrew University Jerusalem Israel
Visiting SRI and Computer Science Stanford University
September 1987[1]

1 Introduction

In the traditional approach to semantics truth values are assigned to sentence types. Moreover, the meaning of linguistic expressions is given through a map which associates with them extralinguistic entities, in such a way that the meaning of a complex expression is derivable from that of its components. By treating equally all tokens of the same expression such an approach makes for an enormous reduction in complextiy. Sometimes modifications are needed: The interpretation of indexicals, expressions such as 'I', 'now' etc., is determined not only by type but by context dependent parameters: a token of 'I' denotes the person uttering it. But these additional parameters can be specified and made explicit, resulting in a picture which is still within the scope of the original conception.

Yet some language games convey meaning in much more complicated ways. They necessitate a radical departure from the denotational style of semantics. These are discourses which involve self-referential applications of semantic predicates, 'true', 'false', as well as modal predicates like 'know' or 'necessary'.

Consider for example the following exchange. Max: "What I am saying at this very moment is nonsense", Moritz: "Yes, what you have just sayed is nonsense". Apparently Max spoke nonsense and Moritz spoke to the point. But Max and Moritz seem to have asserted the same thing: that Max spoke nonsense. Wherefore the difference?

To avoid the vagueness and the conflicting intuitions that go with 'nonsense', let us replace 'nonsense' by 'not true' and recast the puzzle as follows:

line 1 The sentence on line 1 is not true.

line 2 The sentence on line 1 is not true.

[1]The basic ideas underlying this paper were first presented in a conference to the memory of Bar-Hillel held in October 1985 at the University of Boston. Since then the framework has undergone quite a few developments. Various stages have been presented in lectures given at UCLA, Harvard, Princeton, Stanford and UC Irvine (during 1986), at a CSLI conference on the semantics of self-reference (February 1987), at the Pacific Division of the APA meeting (March 1987) and, most recently, in a conference at the University of Texas, Austin. I have benefited from many reactions, observations and discussions of the issues. Among the many to whom thanks are due I would like to mention in particular David Kaplan and Hillary Putnam. I am also indebted to Rohit Parikh for useful comments and for finding a bug in one of the earlier proofs.

If we assume that the sentence on line 1 is true we get a contradiction, because on this assumption what it asserts is true, but what it asserts is that the sentence on line 1 is *not* true. Consequently the sentence on line 1 is not true. But when we write this true conclusion on line 2 we see that we have repeated the very same sentence whose truth we deny. How can we then express this "something" which we feel to be true?

Our puzzle is a reformulation of the Strong Liar designed to bring out the following perspective: The problem raised by the paradox is not the "contradiction in natural language" but the apparent inability to express in the language something we know to be true. This problem is unsolvable if we insist on equating tokens of the same type. But in actual discourse it is solved by making good use of token distinctions.

It is well known, [Montague 1963], that the semantic paradoxes can be reconstructed in various modal frameworks. Indeed, using a knowledge predicate we get an analogous puzzle:

line 1 The sentence on line 1 is not known by Moritz to be true.

Observing this sentence Moritz concludes that he has no knowledge of its truth, because such knowledge would imply that the sentence is true, hence that Moritz does *not* know its truth. Moritz writes his conclusion on line 2:

line 2 The sentence on line 1 is not known by Moritz to be true.

Having deduced this conclusion, Moritz knows it to be true. Thus he knows the truth of the second sentence but not that of the first. But these are occurences of the very same sentence!

With 'necessary' the puzzle is obtained by writing on line 1 "The sentence on line 1 is not necessarily true". Again, this sentence is not necessarily true, because then it would be true, hence *not* necessarily true. This conclusion, written on another line, is necessarily true, because we have just proved it. So the sentence on line 1 is not necessarily true, but its repetition on another line is.

The moral of all these puzzles is simple: In situations of this nature we should assign truth values not to sentence types but to their tokens. The token on line 2 expresses something (fact, statement, proposition – choose your favourite term) altogether different from what is expressed (if anything) by the token on line 1. And, of course, what is expressed depends on the whole network: on the tokens that the sentence refers to and on the tokens that *they* in their turn refer to etc. This is what distinguishes the self referential sentence-token on line 1 from its non self referential brother on line 2.

In this respect, modal predicates like 'know' and 'necessary' are in the same boat with the predicate 'true' and the same remedy is required. It is of course to be expected that the formalisms will differ according to the predicates in question, but the same general framework will underlie them. In the present work we set up the formalism for truth, thereby providing also the framework for the various modalities.

There is also a general perspective from which the present work is relevant to the theory of knowledge. A theory describing how information is expressed through networks of tokens (or, in general, pointers) shows at the same time how knowledge is expressed evaluated and passed on.

We base our formalism on three truth values **T** (*True*), **F** (*False*) and *GAP*. The third value signifies failure to express something which evaluates to either **T** or **F** . The truth values depend on the token's type, i.e., on what it "says", as well as on its place in the network. I shall present a simple general way of specifying such networks and a precisely defined evaluation algorithm for assigning truth values. It assigns the sentence on line 1 the value *GAP*, the sentence on line 2 - the value **T** and it yields similarly intuitive results in other cases.

The concept of token is too narrow for the purpose of a general framework. For we might want to refer to sentences without having them displayed somewhere as tokens. We therefore use a more general concept, that of pointer:

A pointer is any object which is used to point to a sentence type. A token is a special case of pointer – it points to the sentence type of which it is a token.

In our formalism we introduce pointers as a primitive structure, whose interpretation is given by a *pointing function* which associates with every pointer a sentence (or a well formed formula) to which it points. The function can be quite arbitrary, allowing for all possibilities of direct or indirect self reference.

The upshot of this approach is a new kind of semantics in which truth values are assigned to pointers and the usual recursive definition of truth is replaced by an algorithm for evaluating networks. Here is a simple informal illustration of how it works. Let Mary, Marjory, Max and Moritz make the following statements:

Mary: What Moritz says is not true and what Marjory says is not true.

Moritz: What Max says is true.

Marjory: What Moritz says is not true.

Max: Either McX's conjecture is true or what Moritz says is false.

The resulting network is represented in fig 1, where people serve as pointers to the corresponding sentences. The arrows between pointers do not represent the pointing relation (which is a relation between pointers and sentence types) but direct calls in the evaluation procedure, (e.g., a pointer to $A \vee B$ will call pointers to A and to B). The sentence-types associated with the pointers can be read from the diagram though they are not explicitly displayed.

Assume for the sake of illustration that McX's conjecture does not refer back (directly or indirectly) to the utterences of our four speakers. It might involve loops of its own, so the algorithm is applied recursively. Assume that McX is assigned **F**.

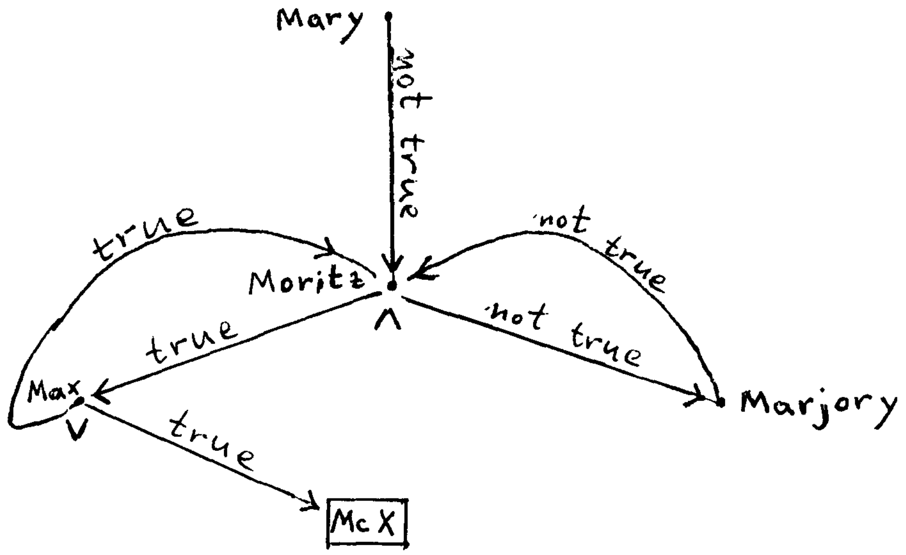
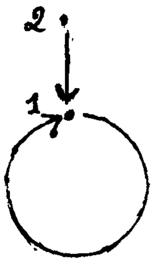


Fig. 1

This leaves Max, Moritz and Marjory in a closed unresolved loop (a concept to be defined formally in the sequel). Both get at this stage the value *GAP* . Then Mary who makes an assertion about Moritz and does not belong to the loop gets a standard (**T** or **F**) value. Since her assertion is true, she gets **T**. Had McX gotten **T** Max would have gotten **T** Moritz and Marjory - *GAP* and Mary - **T** .

In the case of the two line puzzle the network is:



For '*i*' read: 'the sentence token on line *i*'. Right at the beginning we get a closed loop consisting of the pointer 1. It gets the value *GAP*. Then 2 gets the value **T** .

In general, the assignment of *GAP* signifies a decision that the pointers in question fail to evaluate to standard truth values. The ground for this decision is that they constitute a closed loop. In more sophisticated versions of the algorithm, other grounds are considered as well. The main idea is to limit the assignment of *GAP* to a restricted group of "guilty pointers". This leaves us "uninfected pointers" for making, inside the language, the assertions that we want. What is unexpressible in the usual denotational semantics is thus expressible through network evaluation. By making *GAP* into something "positive"—not mere failure but *recognised* failure—we can construct on top of *GAP* instead of falling into it.

Can this idea be formalized and applied to a fully fledged language? We shall see in the coming sections that it can.

[The assignment of truth values to tokens, or in general to pointers, does *not* signify a nominalistic venture of reduction. On the ontological questions concerning propositions, senses, meanings, etc., the proposal is neutral. If we wish, we may regard pointers as channels through which propositions are expressible. We may say, if we wish, that the sentence (token) on line 1 expresses no proposition, or a degenerate proposition, or a circular one. Of course, it is the whole network which determines the proposition and to grasp it completely we have to understand the evaluation procedure. Thus, pointers are in no way a substitute for propositions.]

Most of the works on the semantic paradoxes treat the sentences on lines 1 and 2 in the same way. Parsons [1974] states explicitly that if 'a' and 'α' refer in a clear unproblematic manner to the same sentence then the assertion: "a is true" commits us also to the assertion: "α is true". This is indeed the case in all the models that have been set up along the lines Kripke's proposal [1975] (anticipated by Martin and Woodruff [1975] and rediscovered by Kindt [1979]). These models provide semantics for formal languages, with a distinguished predicate over sentences playing the role of the "truth predicate". The Weak Liar paradox is avoided by admiting some version of truth value gaps ("ordinary" gaps in Kripke's model, sentences with oscilating truth values - in the models of Gupta [1982] and Herzberger [1982]). In many respects this truth predicate simulates the truth predicate of natural languages. But in none of these models can we truly assert that the sentence on line 1 is not true. For the sentence used to make this assertion will share the same fate as the sentence on line 1.

Consequently, we are unable to reconstruct formally what the speakers of language do on the spur of the moment: to realize that by its very nature the sentence on line 1 cannot be true, to assert this fact in the same language and to realize that this second assertion is true. The failure shows that in these attempted modelings we have been unable to capture an essential feature in the functioning of natural language.

In the models just mentioned, if a sentence lacks a (stable) truth value then so does any sentence that says of this sentence that it is not true, or that it is not false, or that it is true, or that it is false. One gets what we call **Black Holes**. This concept and the hierarchy of holes are defined as follows:

S is a *0-hole* if it is a gap and, by induction, S is a *(n+1)-hole* if it is a n-hole and 'S is true' and 'S is false' are gaps. S is a *black hole* if it is a n-hole for all n. By convention, let '*hole*', without prefix, denote 1-holes.

No information concerning the truth value of a hole can be stated directly. [2] As for

[2]We might still convey semantic information indirectly. For if we succeed in asserting " 'S is true' is

black holes, they are semantic untouchables. For no semantic information about them can be conveyed in any way, be it as indirect and across as many layers as you may wish.

In the systems proposed by Kripke, Gupta and Herzberger every gap is a black hole. It can be shown that if truth values depend only on types, then, under some mild assumptions concerning sentence equivalence, the sentence on line 1 is a black hole. Thus, black holes are bound to occur on the most elementary level of language.

The basic (and simplest) version of our proposal eliminates the holes in all situations of finitary type — those in which every pointer calls only finitely many pointers. To this class belong all the tangled loops producible when finitely many people make statements about each other, provided that altogether finitely many statements are involved. In certain situations of infinitary type holes can appear if the language is sufficiently rich. The simpler type of holes can be eliminated by more sophisticated, yet intuitive, versions of the evaluation procedure (but these will be the subject of another paper). It seems that further improvements are possible. How far one can go towards the elimination of holes, in particular black holes, is an intriguing foundational question.

In the broader perspective of natural language "hole like" phenomena are bound to occur. Some discourses take us to the edge of meaning and some tempt us to express the inexpressible, or to think the unthinkable. Holes are perhaps the inevitable price for a powerfull language capable of evolving. But we can at least tidy up the more accessible levels.

I think that a satisfactory treatment of the semantic paradoxes should provide a systematic account of how a network of pointers operates and how various levels of reasoning can be expressed in the same (untyped) language through network evaluation. I do not claim that my particular version is the definitive answer. No single variant will do justice to all intuitions and to all occasions. What I am proposing is an open framework, flexible enough to accomodate a broad range of intuitions.

Previous works which adopted the basic intuition that distinguishes between the sentences on lines 1 and 2, are by Brian Skyrms [1970], [1982] and by Tyler Burge [1979]. They proposed to handle such phenomena with more traditional tools, the first by construing the truth predicate as intensional, the second - as an indexical. These proposals did not yield precisely defined systems. I find the tools which these proposals used to be inadequate for the problem at hand. For reasons of space this issue is analysed only in the expanded version of the paper.

A most recent work on semantic self-reference is that of Barwise and Etchemendy [1987]. It treats the subject within the general framework of situation semantics. The exact relations between their setup and the method proposed here for determining truth values remain to be sorted out.

neither true nor false" then we imply by this that S itself is neither true nor false, the underlying assumption being that if S is true or false then 'S is true' inherits the same truth value. We shall see later how holes which are not 2-holes can arise.

Besides the obvious implications of this proposal for the philosophy of language it bears also on the semantics for computer languages as well as Artificial Intelligence. (The failure of a procedure to return a value may be "recognised" by another procedure, upon which it will declare the value of some pointer to be GAP. When each procedure can point to any other we get networks whose evaluation yields a semantics along the lines proposed here.)

The relevance of our proposal for the various branches of modal logic has been mentioned already. The customary representation of modality as a sentential operant, on a par with connectives, avoids the paradoxes but is quite restrictive; how are we going to construe statements involving the expressions 'knows something' or 'knows everything'? As Morgenstern [1986] observes: "...we cannot formulate such sentences as 'John knows that Bill knows something that he does not know.' Assuming that knowledge about actions is in the form of statements, we also cannot express 'John knows that Bill knows how to fire a gun' unless John himself knows how to fire a gun.". The need for a predicate representation of knowledge is indicated clearly in recent research cf. Thomason [1986].

As we noted above our system has direct implication for the representation of modality by means of predicates over sentence-tokens, or more generally – pointers. Indeed, moves from the theory of truth to the theory of knowledge have been carried out with respect to previous proposals: Asher and Kamp [1986] proposed a model for epistemic modality, based upon the models of Gupta and Herzberger, while Kremer [1986] and Morgenstern [1986] employ in a similar way Kripke's model. (Other recent works motivated by the needs of knowledge theory are by Perlis [1985] and by des Riviers and Levesque [1986].)

2 The Semantics of Pointers

2.1 Pointer Systems

A pointer system for a language $\mathcal{L}$ consists of:

(i) A set $\mathcal{P}$ of objects called *pointers.*

(ii) A mapping $\downarrow$ from $\mathcal{P}$ onto the set of wffs (well formed formulas) of $\mathcal{L}$, associating with every $p \in \mathcal{P}$ a wff $p\downarrow$. We say that *p points to* $p\downarrow$.

(iii) Two functions associating with every $p \in \mathcal{P}$ pointers $p1$ and $p2$ such that: If $p\downarrow = A * B$, where $*$ is a binary connective, then $p1\downarrow = A$ and $p2\downarrow = B$; and if $p\downarrow = \neg A$ then $p1\downarrow = A$ and $p2 = p1$. In all other cases $p1 = p2 = p$.

(In the case of pointers which are tokens $p1$ and $p2$ have natural interpretations: If p is a token of $A * B$ then $p1$ is the part which forms a token of A and $p2$ is the part which forms a token of B.)

We put: $\mathcal{P} = (\mathcal{P}, \downarrow, (\ )1, (\ )2\ )$

[For languages with quantifiers enrich the structure $\mathcal{P}$ as follows: Add a two-place function $(\ )|(\ )$, taking as arguments pointers and terms of $\mathcal{L}$, such that if Q is a quantifier, $p\downarrow = QxA(x)$ and t is a term, then $(p|t)\downarrow = A(t)$.]

There are many natural ways of enriching the structure. We can consider pointers to other linguistic expressions and handle the syntax of the language through them (e.g., with every pointer p to an atomic formula we can associate a pointer $p0$ to the predicate which occurs in this formula). For the purposes of our evaluation procedure the structure as defined here is all we need.

Note that the collection of wffs constitutes trivially a pointer system. Simply define, for every wff A: $A\!\downarrow = A$ and for $A = B * C$ put: $A1 = B$, $A2 = C$ and similarly for negations.

2.2 Pointer Calculus

Assume that $\mathcal{L}$ is based on a vocabulary of individual constants (possibly of various sorts), predicates, function symbols (optional), sentential variables (optional) and the usual sentential connectives. For simplicity we concentrate here on the propositional case. We shall indicate in brackets how the framework extends naturally to languages with quantifiers. All the forthcoming theorems hold for quantified languages as well.

Assume that among the individual constants of $\mathcal{L}$ there are pointer-constants, to be interpreted as pointers to the wffs of $\mathcal{L}$ itself.

Among the predicates there are two distinguished predicates $Tr(\ )$ (for truth) and $Fa(\ )$ (for falsity) taking pointer-constants as arguments.

Tr and Fa are the *truth predicates* (called also semantic predicates). We call wffs of the form $Tr(...)$, $Fa(...)$ *atomic semantic wffs.* All other atomic wffs are called *basic.*

We define a *model* for $\mathcal{L}$ to be a triple: $(\tau,\ \mathcal{P}, \delta)$ such that

(i) τ is a function assigning every basic wff a truth value, which may be either **T** or **F** or GAP.

(ii) $\mathcal{P}$ is a pointer system for $\mathcal{L}$.

(iii) δ is a mapping which associates with every pointer-constant a pointer in $\mathcal{P}$ (the pointer named by the constant).

Let p, q, r, p_1, etc., range over pointers. For simplicity, we assume that their names in $\mathcal{L}$ are 'p', 'q', 'r', 'p_1', etc., i.e. – the same names used in this article.

The atomic semantic wffs are therefore of the form $Tr(p)$ or $Fa(p)$.

For $Tr(p)$ one can read "the value of p is *True*", or "p points to truth" or, in the case of tokens, "the sentence-token p is true"; similarly for $Fa(p)$.

Let A, B, A_1, B_1, ... etc. range over the wffs of $\mathcal{L}$.

$p\!\downarrow\ = A_1, \ldots, A_n$ is a shorthand for: $p\!\downarrow = A_1$ or ... or $p\!\downarrow = A_n$.

Note that all the syntax of our language can be handled within the language, by predicates over pointers, e.g., we can have a predicate $Neg(\ )$, such that $Neg(p)$ is true iff $p\!\downarrow$ is a negation, and similarly for all other syntactic concepts.

2.3 The Network of Pointers

Assume throughout some given model for $\mathcal{L}$.

Definitions *p calls q directly* if either of the following holds:

(i) $p{\downarrow} = \neg A, A * B$ and q is either $p1$ or $p2$.

(ii) $p{\downarrow} = Tr(q), Fa(q)$.

Calls of type (ii) are referred to as (direct) *semantic calls.*

Evidently, $p{\downarrow}$ is basic iff no pointer is called directly by p.

A *network of pointers* is a labeled directed graph whose vertices are pointers, each p is labeled by $p{\downarrow}$ and $(p,\ q)$ is an edge iff p calls q directly.

A *calling path* from p to q is sequence $p_1, \ldots, p_n$, with $n > 1$, $p = p_1, q = p_n$, such that every p_i calls p_{i+1} directly.

p calls q if there is a calling path from p to q.

The network *generated by p* consists of p and all the pointers called by p.

[For languages with quantifiers add to the above definition a third clause: If $p{\downarrow}= QxA(x)$ and t is a constant term then p calls directly $p|t$.]

Directed graphs can be represented by what we call *looped trees*, or *l-trees* – for short. A looped tree is obtainable from a tree by looping back some of its leaves, "looping back" meaning connecting a leaf by a backward going edge to one of its ancestors. This is the analogue of the representation of acyclic graphs by trees. As in the acyclic case, different nodes in the tree may represent the same vertex in the graph. Hence the nodes of the l-tree are to be labeled by the vertices of the graph. There is a simple algorithm for constructing the l-tree representing the network generated by some pointer; for lack of space we omit it. An example of an l-tree is given in Fig. 2 at the end of section 2.4. Note that, except for the leaves, we have only to indicate the major connective or the truth predicate.

Definitions A set of pointers S is a *loop* if $S \neq \emptyset$ and for all $p, q \in S$ there is a calling path in S from p to q.

Note that $\{p\}$ is a loop iff p calls directly itself iff $p{\downarrow} = Tr(p), Fa(p)$.

If $R \subset S$ then *R is closed in S* if every $p \in S$ which is called directly by some pointer in R is in R.

L is a *closed loop in S* if $L \subset S$, L is closed in S and L is a loop.

2.4 The Evaluation Algorithm

A *valuation* of a network is a partial function v which assigns truth values to the pointers in its domain.

$Dom(v)$ is the domain of v.

$v(p)$ is undefined if $p \notin Dom(v)$, in which case we say that *p is unevaluated by v.* The valuation is *total* if all the network pointers are evaluated.

We let v, u, w, v_0,... range over valuations, p, q, r, p_0,... – over pointers.

The evaluation rules are of the form:

$$\text{If} \quad \mathcal{C}(v,p) \quad \text{then} \quad v(p) := value$$

Here $\mathcal{C}(v,p)$ is a condition on the valuation v and the pointer p and $value \in \{\mathbf{T},\mathbf{F},\text{GAP}\}$. "If...then..." is interpreted operationally: *If* $\mathcal{C}(v,p)$ *is satisfied then make the assignment* $v(p) = value$.

Note that 'v' figures here as a program variable which keeps changing during the execution. In '$\mathcal{C}(v,p)$' 'v' denotes the valuation at a certain stage, while in the consequent '$\boldsymbol{v}$' is used to express the assignment statement.

$\mathcal{C}(v,p)$ is called the *enabling condition* of the rule. If this condition is true we say that v *enables the rule for* p, or, for short, that the rule *applies to* p. To apply such a rule means to redefine v by putting: $v(p) = value$. If originally $p \notin Dom(v)$, then this application will extend v, if originally $p \in Dom(v)$ and $v(p) \neq value$ then the application will change an existing value, and if originally $v(p) = value$ it will leave v unchanged.

$\mathbf{T}$ and $\mathbf{F}$ are called *standard values* and we put: $-\mathbf{T} =_{Df} \mathbf{F}$, $-\mathbf{F} =_{Df} \mathbf{T}$.
The rules are divided into standard rules, the jump rule, and the gap rules determining the assignment of GAP.

Standard Rules

For Basic Values:
If $p\downarrow = A$ and A is basic then $v(p) := \tau(A)$

For Negation:
If $p\downarrow = \neg A$ and $v(p1)$ is defined and standard then $v(p) := -v(p1)$.

For Disjunction:
If $p\downarrow = A \vee B$ then

(i) If either $v(p1) = \mathbf{T}$ or $v(p2) = \mathbf{T}$ then $v(p) := \mathbf{T}$
(ii) If $v(p1) = v(p2) = \mathbf{F}$ then $v(p) := \mathbf{F}$.

(If other connectives, say $\wedge$ and $\rightarrow$, are primitives, their standard rules are the obvious analogues of the negation and disjunction rules.)

For the Truth Predicates:
If $p\downarrow = Tr(q)$ and $v(q)$ is defined and standard then $v(p) := v(q)$
If $p\downarrow = Fa(q)$ and $v(q)$ is defined and standard then $v(p) := -v(q)$

Jump Rule

If $p\downarrow = Tr(q), Fa(q)$, $v(q) = GAP$ and $v(p) \neq GAP$ then $v(p) := \mathbf{F}$.
(By $v(p) \neq GAP$ we mean that $v(p)$ is either undefined or standard)

Jump is the rule by which we ascend in the Tarskian hierarchy. If q was assigned GAP then an unevluated p pointing to $Tr(q)$ or to $Fa(q)$ will get $\mathbf{F}$ and if $p = r1$, where $r\downarrow = \neg Tr(q), \neg Fa(q)$ then r will get $\mathbf{T}$. The condition that $v(p) \neq GAP$ is crucial, for it may happen that because of a loop both q and p have been assigned already GAP, in which case we cannot assign p a standard value.

The following are the gap rules.

Simple-Gap Rule

If v is defined for all pointers called directly by p and none of the preceding rules applies to p, then $v(p) := GAP$.

Closed-Loop Rule

If S is a closed loop in the set of all pointers unevaluated by v and none of the preceding rules applies to to any p in S, then $v(p) := GAP$ for all $p \in S$.

An application of this rule means, by definition, the assigning of GAP to *all* pointers in S, we cannot leave some of them unassigned. This is also the case in the next and last rule.

Give-Up Rule

If the set of unevaluated pointers is not empty and none of the preceding rules apply to any of its members, then $v(p) := GAP$ for all unevaluated p.

[For languages with quantifiers add standard rules for quantified sentences by treating existential and universal sentences as infinite disjunctions and conjunctions.]

Kleene's strong 3-valued truth tables are implied by the standard rules for connectives and the simple-gap rule. The standard rules can however be replaced by other schemes, for example – supervaluation schemes which will cause pointers to tautologies to have the value **T**. The whole setup is modular in that we can change the standard rules without changing any of the rest.

Given a model and using the empty valuation as a starting point, we can apply repeatedly the rules. One of our theorems implies that eventually we shall reach a total valuation which is closed under the rules: they become true statements when "if... then..." is interpreted as a material implication and ':=' is replaced by '='. Moreover this is true for a very wide class of starting points. Another result states that the final valuation depends only on the starting point, not on the choice of rules to be applied at each stage.

EXAMPLE The l-tree given below represents a network whose pointing function is as follows:

$$\begin{aligned}
o\!\downarrow &= \neg Tr(p)\\
p\!\downarrow &= Tr(q) \wedge Fa(r)\\
q\!\downarrow &= Fa(s) \vee Tr(p1)\\
s\!\downarrow &= Tr(q1) \wedge Tr(s)\\
r\!\downarrow &= Tr(s) \wedge (X \vee Tr(p2))
\end{aligned}$$

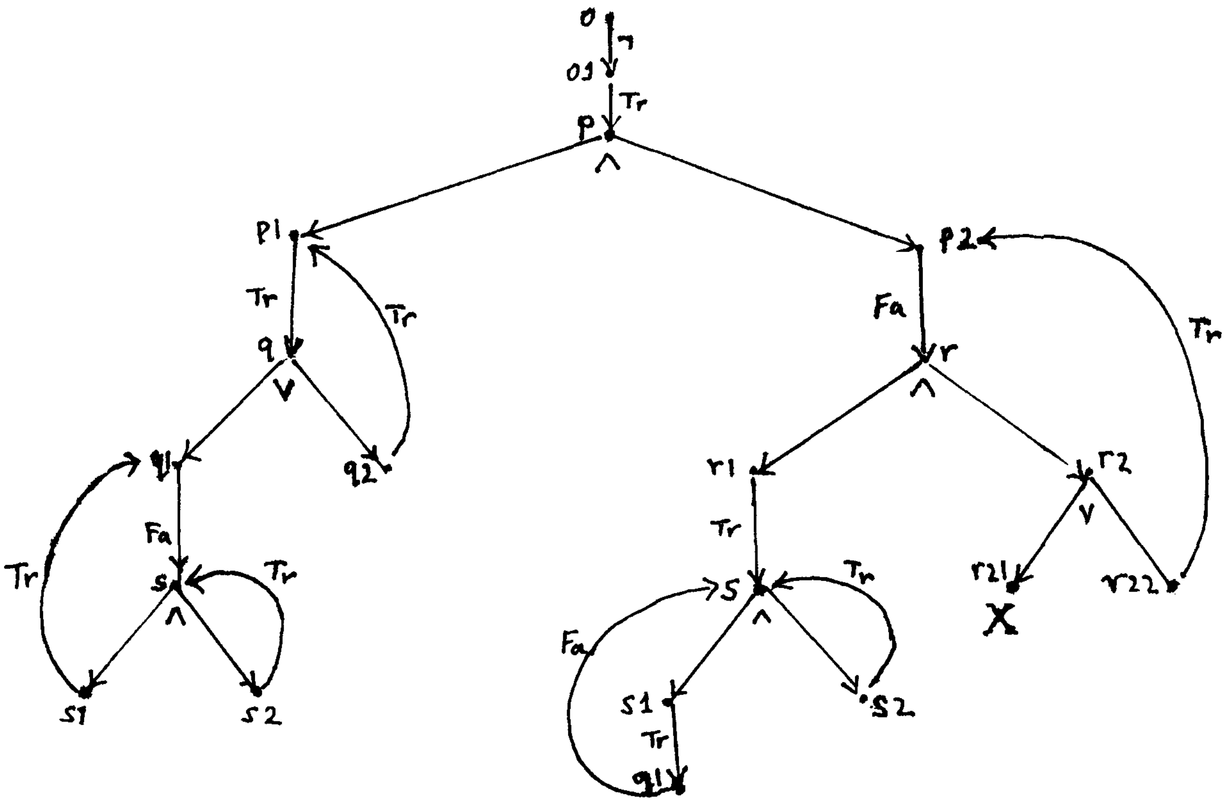


Fig. 2

For $r(X) = \mathbf{F}$, the evaluation proceedure yields:

$v(q1) = v(s1) = v(s) = v(s2) = GAP$ (closed loop rule)
$v(r1) = \mathbf{F}$ (Jump rule)
$v(r) = \mathbf{F}$ (standard rule for $\wedge$)
$v(r21) = \mathbf{F}$ (basic values rule)
$v(p2) = \mathbf{T}$ (standard rule for $Fa(\ )$)
$v(r22) = \mathbf{T}$ (standard rule for $Tr(\ )$)
$v(r2) = \mathbf{T}$ (standard rule for $\vee$)
$v(q) = v(q2) = v(p1) = GAP$ (closed loop rule)
$v(p) = GAP$ (simple gap rule)
$v(o1) = \mathbf{F}$ (Jump rule)
$v(o) = \mathbf{T}$ (standard rule for $\neg$)

2.5 Evaluation Sequences and Self-Supporting Valuations

We now provide a formal analysis. The proofs of the theorems are omitted for reasons of space. They will appear in the expanded version.

Definition For a given valuation v the *derived valuation*, $v^{\#}$, is the valuation obtained by one concurrent application of all the rules which are enabled by v and by deleting from $Dom(v)$ the pointers for which no rule is enabled. Formally:

$$\begin{aligned} Dom(v^{\#}) &= \{p : \ v \text{ enables some rule for } p\ \} \\ v^{\#}(p) &= \text{ the value assigned to } p \text{ by the rule which } v \text{ enables for it.} \end{aligned}$$

This is legitimate because for each p at most one rule is enabled.

Note that the derivative operation is not monotone: $v \subset u$ does *not* imply $v^{\#} \subset u^{\#}$.

Define a *successor of* v to be any valuation u obtained from v by applying to some pointers the rules enabled for them (including possibly deletions from $Dom(v)$ of some pointers to which no rule applies) and leaving the rest of v intact. Formally, u is a successor of v if: $Dom(v) \cap Dom(v^{\#}) \subset Dom(u) \subset Dom(v) \cup Dom(v^{\#})$ and for all $p \in Dom(u)$:

(i) Either $u(p) = v(p)$ or $u(p) = v^{\#}(p)$ and

(ii) If $u(p) = v^{\#}(p)$ and the value is obtained by an application of the closed-loop or give-up rule, then all the pointers which are assigned GAP by this application are in $Dom(u)$.

Definition An *evaluation sequence* is a well ordered sequence $v_0, v_1, \ldots, v_\alpha, \ldots, v_\lambda$, with a last member v_λ such that: (i) For all $\alpha < \lambda$, $v_{\alpha+1}$ is a successor of v_α. (ii) $\alpha \leq \alpha'$ implies $v_\alpha \subset v_{\alpha'}$. (iii) For α a limit ordinal $v_\alpha = \bigcup_{\gamma<\alpha} v_\gamma$.

We call v_0 the *starting point* and say that the evaluation sequence *begins with* v_0 and *ends with* v_λ.

Note that the sequence is required to be ascending. But we shall see that for well behaved starting points any choice of successor will constitute an extension and this will be preserved throughout the evaluation sequence.

Every non-total v enables some rule for some unevaluated pointer (the give-up rule is enabled if no other rule is). Hence it has a proper successor-extension. This easily implies:

Proposition 1 *For every v there is an evaluation sequence beginning with v and ending with a total valuation.*

Definition

A valuation v is *supported* on p if $p \in Dom(v)$ and $v(p) = v^{\#}(p)$.

A valuation is *self-supporting* if it is supported on all pointers in its domain (or, equivalently, if $v \subset v^{\#}$).

A valuation v is *complete* if $v = v^{\#}$.

Note: If $p\downarrow = Tr(p)$ then any valuation defined on p is supported on p. But if $p\downarrow = Fa(p), \neg Tr(p)$ then v is supported on p iff $v(p) = GAP$. The distinction between the Truth-Teller and the Liar is thus brought out.

It is easily seen that a valuation is complete iff it is self-supporting and total. Also if v is self-supporting then any successor of v extends v (follows trivially from $v \subset v^{\#}$ and the definition of successor).

Proposition 2 *(i) If v is supported on p and u extends v then u is supported on p. (ii) If u is a successor of v which extends v then u is supported on all $p \in Dom(u) - Dom(v)$. (iii) If v is self-supporting then any successor of v is self-supporting.*

By a *chain* of valuations we mean a family of valuations which is totally ordered by inclusion.

Proposition 3 *A union of a chain of self-supporting valuations is self-supporting.*

Using the last two propositions one shows:

Theorem 1 *Let v_0 be any self-supporting valuation. Construct a sequence as follows: If v_α is defined and has a successor different from it, choose as $v_{\alpha+1}$ any such successor and, for limit ordinals β, if the v_γ are defined for all $\gamma < \beta$ put: $v_\beta = \bigcup_{\gamma<\beta} v_\gamma$.*

Then this is an evaluation sequence, all v_α are self-supporting and the last one is a complete valuation.

The next theorem guarantees that the end results do not depend on the order of applying the rules. The proof uses more delicate arguments than those used for the previous theorems. Note that we cannot appeal to some fixpoint argument because we are not in a monotone situation.

Let S be a non-empty set of pointers unevaluated by v. Say that S is a *gap set for v* if it is the set of all pointers which are assigned GAP by a single application of a gap rule to v. We have 3 kinds of gap sets: A *simple-gap set* is a singleton consisting of an unevaluated pointer to which the simple-gap rule applies. A a *closed-loop set* is a closed loop in the unevaluated pointers to which the closed-loop rule applies. A *give-up set* consists of all unevaluated pointers when the give-up rule applies. (The simple-gap rule may apply also to an evaluated pointer, but for a single-gap set we require it to be unevaluated.)

Theorem 2 *Let $u_0, u_1, \ldots$ and $v_0, v_1, \ldots$ be evaluation sequences ending with the total valuations u and v, respectively. If $u_0 = v_0$, then $u = v$, and the simple-gap sets closed-loop sets and give-up sets (if any) of the two sequences are the same.*

[In the proof one shows by induction on α that (i) $u_\alpha \subset v$ and (ii) Any gap set for u_α is also a gap set of the same kind for some v_β]

3 Basic Results

Theorem 3 *Let v be any complete valuation. If $p\!\downarrow = q\!\downarrow$ then either $v(p) = v(q)$ or one of $v(p)$, $v(q)$ is GAP.*

The proof is by unduction on $p\!\downarrow$.

Let σ be a mapping of all the atomic wffs of $\mathcal{L}$ (semantic and non-semantic) into {T , F}. Regard σ as a classical model for $\mathcal{L}$ in which $Tr(\ )$ and $Fa(\ )$ are treated like any other predicates. For a complete valuation v, we say that σ is *correlated with* v if for all p such that $p\!\downarrow$ is atomic, if $v(p)$ is standard then $\sigma(p\!\downarrow) = v(p)$.

The previous theorem implies that every complete valuation has a correlated classical model.

Proposition 4 *Let σ be a classical model correlated with v. Extend σ to all wffs by the usual rules for satisfaction. Then, for all p, if $v(p)$ is standard then $\sigma(p\!\downarrow) = v(p)$.*

Using Proposition 4 one can get a strengthening of Theorem 3:

Theorem 3* *If $p\!\downarrow$ and $q\!\downarrow$ are logically equivalent then, for every complete valuation v, either $v(p) = v(q)$ or one of $v(p)$, $v(q)$ is GAP. Also if $p\!\downarrow$ is logically valid $v(p)$ is either* **T** *or GAP.*

(Here "logical equivalence" is equivalence via the usual logic rules, with the truth predicates treated like any other.)

For a given model M, the *valuation determined by M* is the complete valuation obtained via an evaluation sequence with an empty initialization. It is not difficult to see that when evaluating any pointer we need to consider only the network generated by it. This easily implies:

Theorem 4 *If v is determined by some model, $p\!\downarrow = Tr(q), Fa(q)$ and q does not call p then $v(p)$ is standard.*

Definition A network is *locally finite* if every pointer calls only finitely many pointers.

Theorem 5 *If the network is locally finite no (partial) valuation enables the give-up rule.*

The proof is by observing that any finite set of pointers either contains as a subset a closed loop, or has a member which does not call any pointer of the set.

Hence, for locally finite networks, we can delete the give-up rule.

Define a *subpointer* of p to be either p, or pi, $i = 1, 2$ or, recursively, any subpointer of pi. Evidently, for each subformula of $p\!\downarrow$ there is exactly one subpointer of p pointing to it.

For the next theorem we assume that for every formula there are infinitely many pointers to it having disjoint sets of subpointers.

Theorem 6 *If v is a valuation determined by a locally finite model, then:*
(i) Every wff A has a pointer, p, to it such that $v(p)$ is standard.
(ii) If $v(p) = GAP$, there exists q pointing to $\neg Tr(p) \wedge \neg Fa(p)$ such that $v(q) =$ **T**.

The theorem implies that in the locally finte case we can always assert truly that a gap is a gap, thus there are no holes. Local finiteness is a sufficient but not necessary condition for absence of holes.

When quantifiers over pointers are available, local finiteness is not satisfied because infinitely many pointers can be called through a quantifier. But if every quntification can be reduced to finite conjunctions or disjunctions the conclusions of the theorem will hold (provided that after carrying all the reductions the network is locally finite). This means that we can use quantification of the form $\forall x(A(x) \rightarrow B(x))$ where $A(x)$ is a wff not containing semantic predicates which is satisfied by finitely many pointers. Therefore assertions of the form: "Everything that McX said is ..." do not give rise to holes.

Some Postponed Topics

With unbounded quantification holes can arise provided that the language is sufficiently rich. For example, consider a formalisation of something like "Every pointer which points to me and which says that I am true is not true". This may yiels a hole, but not a black hole (if Max asserts of this pointer that it is not true he will get GAP, but then Moritz can assert that Max's assertion is not true and get **T**). There are more sophisticated versions of the evaluation algorithm which prevent holes of this and related forms. Their discussion is postponed to the next paper. Also postponed to the next paper is the definition of the super-valuation version of the algorithm.

It is not difficult to see that if we are forced to employ the give-up rule then there are black holes. Roughly speaking, the give-up rule is enabled due to the presence of infinite descending branches in the l-tree (this is necessary but not sufficient for give-up). I think that in the propositional case we are justified in not considering models involving such infinite descent. But for languages which are sufficient expressive with respect to pointers and which contain arithmetic such chains are producible by Gödel's techniques. These and some ideas of dealing with such phenomena will be discussed in another paper.

Tarski's hierarchy can be reconstructed within our framework. The idea is that the application of the Jump rule moves us up to a higher major level. Each major levels can be further stratifed by counting the application depth of the standard rules for the truth predicates and the gap rules. This chapter has been omitted for reasons of space and will appear in the expanded version.

REFERNCES

(*TARK* is an abreviation for: *Theoretical Aspects of Reasoning About Knowledge* Proceedings of the 1986 conference, J. Halpern ed. Morgan Kaufman publisher.)

N. Asher and H. Kamp 1986 "The Knowers Paradox and Representational Theories of Attitudes" *TARK* pp. 131 - 148.

J. Barwise and J. Etchemendy 1987 *The Liar, an Essay in Truth and Circularity* Oxford University Press.

T. Burge 1979 "Semantical Paradox" *The Journal of Philosophy 76* pp. 169 - 198.

J. des Rivieres and H. Levesque 1986 " The Consistancy of Syntactical Treatment of Knowledge" *TARK* pp. 115 - 130.

A. Gupta 1982 "Truth and Paradox" *Journal of Philosophical Logic 11* pp. 1 - 60.

H. Herzberger 1982 "Notes on Naive Semantics" *Journal of Philosophical Logic 11* pp. 61 - 102.

Kindt 1979 "Introduction of the truth predicates into first order languages" in Formal semantics and pragmatics for natural languages ed. Guenthner and Schmidt, Reidel.

M. Kremmer 1986 *Logic and Truth* Ph.D. Dissertation, University of Pitsburg.

S. Kripke 1975 "Outline of a Theory of Truth" *Journal of Philosophy 72* pp. 690 - 716.

R. Martin and P. Woodruff 1975 "On Representing "True-in-L" in L" *Philosophia 5* pp. 213 - 217.

R. Montague 1963 "Syntactical Treatments of Modality, with Corollaries on Reflexion Principles and Finite Axiomatizability" *Acta Philosophica Fennica 16* pp. 153 - 167.

L. Morgenstern 1986 "A First Order Theory of Planning, Knowledge, and Action" *TARK* pp. 99 - 115.

C. Parsons 1974 "The Liar Paradox" *Journal of Philosophical Logic 3* pp. 381 - 412.

D. Perlis 1985 "Languages with Self-Reference I: Foundations" *Artificial Intelligence 25* pp. 301 - 322.

B. Skyrms 1970 "Return of the Liar; Three-Valued Logic and the Nature of Truth" *American Philosophical Quarterly 7* pp. 153 - 161

B. Skyrms 1982 "Intensional Aspects of Semantical Self Reference" notes, reprinted in *Recent Essays on Truth and the Liar Paradox* ed. Martin 1984 Oxford University Press.

R. Thomason 1986 "Paradoxes and Semantic Representation" *TARK* pp. 225 - 239