Automated Deduction and Artificial Intelligence

Neil Tennant

As a central normative component of cognitive science, computational logic should direct its interests first and foremost to the design of ideal inference engines for artificial intelligence. The computational logician wants the computer to solve deductive problems. A deductive problem is of the form X ?– A, where X is a finite set of sentences (the premisses) and A is a sentence (the conclusion). If the computational logician succeeds, the resulting programs can be used as components in the design of an artificial intelligence. For in designing an artificial intelligence we want a *cybernetic theorizer*: we want cybernetic development of the logical consequences of sets of mathematical axioms; cybernetic unfolding of the predictive content of our scientific hypotheses; and cybernetic revision of those theories in the light of contradictions discovered, whether internally or with recalcitrant observation. In short, wherever logical reasoning features in our own framing, testing and revision of theories, we should aim to have computational surrogates for that exercise of our own logical intelligence. I say this in full awareness of the undecidability of many logics, and of theories based on them. What we need is *eine Einstellung zur Seele als Maschine*, an attitude that concedes the possibility of cybernetic emulation of our logical abilities.

Computational logic can take two forms. First, one could devise logic programs that would prompt the human user interactively when the human user is seeking a proof of a given argument. In this rôle the computer with its program is no more than a desk-mate, relieving the human user of some of the formal drudgery involved in precise rule applications, and reminding the human user of the full gamut of permissible moves that might be made at any stage with the deductive materials at hand. It would still be up to the human user to make the strategic decisions as to where to look next for an unbroken path through the logical foothills. This is called *interactive* theorem-proving, and I am not interested in it.

What interests me rather is *automated* theorem proving. On this approach, the computer is programmed to carry out the whole proof search entirely on its own, following the search algorithm already embodied in the program. Such programs can be logically complete only if the logical system (or the theory within which one is working) is *decidable*, and not merely itself complete. With an undecidable system, every such (terminating) program for problem-focused proof search will be incomplete. But so too would be

R. Casati, B. Smith, G. White (eds.) Philosophy and the Cognitive Sciences 273-286

that of any human mathematician for the system concerned; for the mathematician is just another machine. So even with respect to an undecidable system one could emulate the abilities of the human logician or mathematician. Even if we cannot match each and every human reasoner within one all-encompassing program, then at least we can furnish for each and every human reasoner a matching program. The aim, then, is to make an automatic theorem-proving program that could take its place within the human mathematical community, and earn its colours by passing the sort of Turing Test that mathematicians constantly apply to each other in sustaining their sense of community.

As we know from applications of computers in other areas, their great advantage is their astonishing speed and their formidable accuracy. They can take the drudge out of the logician's task of formal pattern-matching when invoking axiom schemata and applying rules of inference. They can also make sure that no logical slips are made – that everything done so far has been done correctly, according to the formal rules of the system concerned. Computers keep our logical toils honest. They force us to abide by the rules we have laid down. These rules specify what inferential moves we may make; and what moves the doubting audience must accept as licit. There is astonishing hardness in the silicon must.

Computational logic, then, even conceived as a part of cognitive science, is a normative enterprise. One is seeking a model of competence, not of performance. The methodology is *a priori*, and involves expert introspection. The challenge is to identify, formulate and implement constraints governing search for proofs within given theories.

In a project like this, one has to get three things right:

- a theoretically well-informed choice of logic (where the theoretical considerations can concern such matters as computational complexity, philosophical grounding, adequacy for science and mathematics, etc.)
- a decision algorithm that is no more complex than it has to be for the logic in question (where the algorithm can be one that searches for proofs and/or counterexamples, and where the proofs can be in various formats)
- an implementation of that algorithm that is as efficient as possible (where the implementation should be suitably general and uniform across all possible input problems)

If you want a novice to become a good architect, let her start with Lego. If you want to build a super-sophisticated deductive system, start with simple p's and q's. I shall now describe some theory developed to devise automated proof search for certain decidable propositional logics: logics deserving to be called systems of constructive and relevant reasoning. (A *propositional* logic deals only with logical operators that form sentences from one or more sentences. The most familiar of these are $\sim$ (negation), & (conjunction),

$\vee$ (disjunction) and $\supset$ (implication). A logic is *decidable* when there is an effective method for deciding, of any finite set of premisses and a conclusion, whether the latter can be deduced from the former within that logic.)

Deep and interesting problems emerge for the automation of proof search even with these simple materials: problems the shape of whose general theoretical solution one must discover and appreciate if one is to hope for any progress later with more complicated systems. The computational explosion involved in proof search, even in decidable propositional logics, is awesome. One has to learn how to tame it before tackling the quantifiers, which take one into the realm of the undecidable.

The computational logician is interested not just in the question of whether a logic is decidable, but also in the question whether its decision problem is *tractable*. Every tractable problem is decidable, but not conversely. Tractable problems are conventionally taken to be those that can be solved in polynomial time. Thus, the decision problem for a logic if there is a polynomial function $f(\cdot)$ such that for any problem X ?– A of length n, it could be decided within $f(n)$ steps (or units of computing time) whether there was a proof of the conclusion A from the set X of premisses within that logic. Alas, none of the usual propositional logics turns out to be tractable in this sense.

Classical logic C properly contains *intuitionistic* logic I, which properly contains *minimal* logic M. M consists of just the introduction and elimination rules for the logical operators. I has in addition the absurdity rule, allowing us to infer anything from a contradiction. C goes even further by adding one of the classical rules of negation: the law of double negation, or classical reductio, or the rule of dilemma, or the law of excluded middle. Within I, M has an overlapping neighbour: the system IR of intuitionistic relevant logic. From M there is an easy theoretical transition to IR. This system, I have argued elsewhere, has strong meaning-theoretic, epistemological and methodological claims to adequacy and correctness. I maintain it is the right logic. Like M, it lacks the absurdity rule; but unlike M, it contains disjunctive syllogism and *lacks* the inference $A, \sim A \vdash \sim B$.

Because of its extra negation rules, C has the tamest decision problem of all: although it is not tractable in the sense just explained, it is nevertheless *NP-complete*. What does it mean when one says that a computational problem class (such as deciding whether a given deductive problem admits of proof) is NP-complete? It means that it can be solved in non-deterministic polynomial time (that is, it is NP-easy); and that any other problem class that can be so solved can be transformed into the one at hand in (deterministic) polynomial time (that is, it is NP-hard). This in turn raises definitional demands regarding non-deterministic polynomial time and (deterministic) polynomial time. A problem class can be solved in *non-deterministic polynomial time* just in case there is an algorithm α for solving it, and a polynomial function $f(\cdot)$, such that for any input problem π of length n, the search tree generated by the branching (choice) points of

the algorithm α applied to the problem π contains a solution at the end of a branch of length less than $f(n)$. An intuitive way of thinking of this is that if, in the execution of the algorithm α (which requires us at branching points to make single choices from a range of possible alternatives) we were so it lucky as always to choose "correctly" – and thereby solve the problem as quickly as possible – then indeed we would do so in no more than $f(n)$ units of time.

P is the class of problems (more exactly: problem classes) that can be computed in polynomial time. NP is the class of problem classes that can be computed in non-deterministic polynomial time. It is an open problem whether P = NP. The orthodox conjecture is that P is properly contained in NP; for no-one has ever provided, for any one of the hundreds of problem classes now known to be NP-complete, an algorithm that runs in polynomial time.

C is NP-complete because it provides a very replete set of proof methods. Both I and M, by contrast, are PSPACE-complete. This means that the amount of memory (rather than the number of units of time) needed for the relevant computations is bounded above by a polynomial function. Computations in general need more time than memory, for it takes time to exploit memory storage. So PSPACE-completeness is likely to be worse than P(time)-completeness; indeed, likely to be worse even than NP-completeness.

Let us look more closely at the problem of proof search for systems of constructive and relevant reasoning. For systems such as M, I and IR, there are no such results as conjunctive normal form or disjunctive normal form theorems. The de Morgan laws and dualities break down. Double negations cannot in general be eliminated. One cannot simplify a problem by normal-forming of formulae, or pre-processing its premisses and conclusion before embarking in earnest on proof-search with the simplified forms. Thus the so-called resolution method in automated theorem proving, insofar as it relies on such normal-forming of formulae, is confined to classical logic.

The computational credentials of IR are as compelling as its philosophical and metamathematical ones. By this I mean that its exploitation of the notion of relevance will not make it strictly more difficult to find proofs than it already is in intuitionistic logic. My method of relevantising a system of logic will produce a relevant system whose decision problem is no more complex than the decision problem of the parent system. The method of relevantising the system I of intuitionistic logic to get the system IR also produces, in a exactly similar fashion, the system CR of classical relevant logic from the system C of classical logic. Both IR and CR are decidable, and have decision problems no more complex than those of I and of C respectively.

By contrast, the propositional relevance logic R of Anderson and Belnap is undecidable. Its well-known decidable fragment LR, obtained by dropping the distributivity axiom, has an awesomely complex decision prob-

lem: at best ESPACE hard, at worst space-hard in a function that is primitive recursive in the generalised Ackerman exponential. From NP to ESPACE or worse, courtesy of "relevance"! – so much the worse, then, for this brand of relevance.

These complexity results for LR (and the undecidability of R itself) bring out an important tension between one motivation for studying relevance, and the Anderson-Belnap paradigm for treating it. The *motivation* is that a proof system obeying some constraint of relevance of assumptions to conclusion would admit of faster proof search. (The untutored idea is that a suitable relevance constraint will somehow narrow the search space.) The Anderson-Belnap *paradigm* for treating relevance is their family of systems, clustering around R, that enjoy unrestricted transitivity of proof, eschew disjunctive syllogism, and involve so-called "intensional" connectives. If the motivation and the untutored idea are sound, we must conclude, in the light of the complexity results, that the Anderson-Belnap approach to relevance is not. The possibility remains that some other characterization of relevance will yield the sought reduction in complexity of proof search (or, at least, no increase). Here is an opportunity for computational concerns to inform philosophical preferences for one system of relevance logic over another.

No-one interested in the logical reconstruction of mathematics should undertake it in the relevance logic of Anderson and Belnap. For it forces one to give up disjunctive syllogism – because it holds on to unrestricted transitivity of deduction. Those who would "relevantise" mathematics in R have to re-derive every well-known mathematical theorem from the accepted axioms. They have no general metatheorem to the effect that if a result holds classically, then there is a relevant proof of it.

By contrast with R, we have metatheorems, for the systems IR and CR of relevance logic, guaranteeing the relevantisability of consistent mathematical theories:

Theorem 1 *Any proof in I [C] of a conclusion A from a set X of premisses can be transformed into a proof in IR [CR] of either A or* Λ *(absurdity) from (some subset of) X.*

This guarantees epistemic gain. On relevantising any classical or intuitionistic proof, one obtains either a proof of the sought conclusion from the set X of original premisses, or a proof of that conclusion from some proper subset of X, or a proof that (some subset of) X is inconsistent. We may sum up by saying:

- we may relevantise without loss on consistent sets of premisses
- we may relevantise without loss on logical truths (i.e., on the empty set of premisses)
- we may relevantise without loss on inconsistent sets of premisses

So on the assumption that mathematics is consistent, we are assured that every mathematical theorem (classical or intuitionistic) admits of the corresponding kind of relevant proof. And if mathematics is not consistent, we are assured further that we shall be able to prove this relevantly. We are assured, moreover, that every logical truth can be proved in relevant logic. Finally, we are assured that the hypothetico-deductive method of science, which involves the logical pursuit of inconsistencies between hypotheses and observational evidence, can be carried out using relevant logic.

The systems IR and CR can do everything that any intuitionist or classicist, respectively, could wish their logic to do. Furthermore, we have now the prospect of exploiting the relevance relation (as these systems explicate it) so as to speed up proof-search – or, at the very least, not slow it down. Propositional CR should be no harder than NP; propositional IR no harder than PSPACE. And this is indeed the case.

To summarise, then, we have the following contrasts between the Anderson-Belnap approach to relevance and the approach that I favour:

- They can relevantise mathematics only piecemeal. By contrast, there are metatheorems guaranteeing the relevantisability of mathematics in IR and in CR.
- They keep unrestricted transitivity of deduction, and abandon disjunctive syllogism. By contrast, in IR and CR the transitivity of deduction is controlled in an epistemically gainful way, and disjunctive syllogism is retained.
- Their propositional logics are either undecidable or have decision problems of awesome complexity, compared to those of their parent systems. By contrast, my propositional logics lead to no increase in the complexity of the decision problem.

A central methodological question in connection with any logic we may try to treat computationally is: Do we search for proofs by brute methods or refined ones? By machine-driven merry dance, as with resolution or model elimination methods – or by somehow emulating or simulating competent human interests and methods?

Considered as part of cognitive science, computational logic faces a double challenge. First, one still has the old challenge of using computers as prosthetic devices. That is, one programs computers to find solutions to difficult problems faster and much more accurately than the unaided human mind. One can meet this challenge without any concern for naturalness, simplicity, elegance or "human-like" features of one's search algorithms, except insofar as these features might conduce anyway to greater speed and efficiency in the execution of the programs on the available hardware.

As part of cognitive science, computational logic faces a new challenge: that of programming the machines to *emulate* human reasoning by *simulating* it. That is, one tries to design algorithms for proof search, to be

executed by the machines, that are as isomorphic as possible to whatever collection of methods is employed by competent human reasoners in search of suasive arguments. To face this new challenge, as a cognitive scientist, is to give hostage to fortune in the competition to design proof-finders that do their work simply as fast as possible. For the available hardware, because of its radically different physical construction from the human brain, might be much better at some tasks than the human being; and, correlatively, might be much worse at others.

The human brain, as a product of natural selection, has highly evolved abilities to recall and match visual patterns. This ability no doubt features crucially in higher order human competence in schematic reasoning – at least on reasonably short problems. Likewise, we are able to remember past attempted moves and their outcomes when solving a problem. (And the word 'remember', remember, is ambiguous between record and recall.) Our relatively effortless memory of what we have recently done enables us to learn rapidly from past mistakes.

In both these respects we are, arguably, somewhat different from today's programmable hardware. Depending on one's programming language, there may be a disproportionately higher cost, in the case of a machine, associated with recording and consulting results of recently past computations. And pattern-matching and other forms of associative learning may be severely hampered by the physical design of the hardware. We may have to wait for an engineering revolution in the design of neural networks before our prosthetic devices' profile of relative competences begins to match our own. Just having the theoretical assurance that any Turing machine can be modelled by one of today's digital computers offers no comfort to the cognitive scientist endeavouring to use those computers to simulate our own range of competences in a way that would be real-time faithful.

With the human profile of competence possibly drastically skewed with respect to that of the digital computer, it may turn out that unnatural algorithms can be executed faster than the natural ones that reflect specifically human techniques, abilities, interests and methods. This must constantly be borne in mind when assessing the models of reasoning, or of proof search, offered by computational logic as a branch of cognitive science. Only then can we properly compare the achievements of researchers who employ any brute-force or machine-friendly method, with those of researchers who want to "make the machine think the way we do". If the latter can come close to matching the achievements of the former as far as execution times are concerned, that would already be cause for considerable satisfaction. (Especially when one considers how late has been the entry of "natural deduction"-minded logicians into the field of computational logic.) But there is the exciting prospect also of achieving the added benefit of, say, finding the very proofs that human beings would find, by following methods that human beings themselves deploy. So I would venture to suggest that, in addition to execution times, one consider the nature of the process and features of the

output – in particular, the length and structure of a natural deduction – before entering a decision as to which computational logic program is optimal as a model of human deductive reasoning.

Deductive logic is the centrepiece of any model of (ideal) cognition and reasoning. One's metaphysical stance can influence choice of methodology: e.g. choice of syntactic, proof-theoretic methods over semantic, model-theoretic methods which in their full extent can deal with infinitary objects. A cognitive scientist whose metaphysical position is basically materialist, and who is impressed by the finitude of the neural network, will incline towards models of cognition and reasoning that involve effective transformation of finitary representations.

Now from the standpoint of one interested in human cognitive competence – and in particular the ability to reason logically – "natural proof search" methods seem strikingly underdeveloped. Computational logic needs to explore the algorithmic gains in efficiency on offer from over five decades of proof theory. The kind of proof theory I have in mind here is what might be called intra-systematic proof theory. Its main concern is to achieve a thorough understanding of what a given proof system is like "from the inside". It studies the structure, in the system, of proofs in normal form. The system is characterized by its rules of inference and by the way steps according to them can be patterned so as to form proofs.

It has been the central concern of the work reported on here[1] to explore what proof theory can offer computational logic. Successful proof-search can be very fast, when it is guided by constraints deriving from a deeper understanding of the structure of proofs in normal form. There are also some unexpected benefits for proof theory in confronting the exigencies of computation. The main one is a hybrid system of proof that can be characterised as midway between a Gentzen sequent system and a Prawitz-style natural deduction system.

Another closely connected concern is to develop methods to deal with propositional logic that will generalise smoothly to first order logic. We wish emphatically to avoid any hacker's devices that will not survive the lift to first order. We want, as far as possible, to keep our algorithmic principles uniform across the whole class of input problems. It is this concern that gives us further reason to explore systems of natural deduction, and attack the problem of how to find or generate proofs as suitably structured patterns of sentences.

When searching for proofs we are seeking to construct tree-like arrays of sentences satisfying local or global constraints on their syntactic patterning. Intelligent – that is, highly constrained – search would be best secured by applying the knowledge we have from proof theory about the shape of proofs in normal form, and the transformations that convert proofs into normal form.

[1] N. Tennant 1992.

Systems of natural deduction offer a rich variety of what might be called *completeness-conserving constraints.* The main theoretical investigations to be presented below concern the existence of various kinds of normal forms for proofs of given problems. Suppose P is an effectively decidable property of (X, A) and F is an effectively decidable and non-trivial – that is, constraining – property of proofs. Then what I shall call a *PF-normal form theorem* is a result of the following form:

> for all X and for all A, if $P(X, A)$ then for all proofs Π of A from any subset Y of X, there is a proof Σ of A from some subset Z of Y such that $F(Z, A, \Sigma)$.

Such a theorem gives constraining heuristic guidance in the search for proofs for the problem X ?– A. One checks whether $P(X, A)$. If so, then one confines one's search to proofs with property F. Ordinary *normalization* theorems are special cases of results of the above form:

> for all X, for all A, and for all proofs Π of A from any subset Y of X, there is a proof Σ of A from some subset Z of Y such that Σ is in normal form.

Note that the precondition P in their statement is trivial, and F is the property of normality as usually understood (that is, "not containing any maximal formula occurrence – an occurrence standing as the conclusion of an introduction rule and as the major premiss of the corresponding elimination rule"). Naturally we exploit this conventional normal form theorem, in that we seek only proofs in such conventional normal form. But we supplement this obvious focusing of our search with further PF-normal form theorems, for non-trivial preconditions P, tailored for service in computational logic.

Another special case of PF-normal form theorems is where the relational property F is restricted to the arguments Z and A and is *persistent*, in the sense that if $F(Z, A)$ and Z is a subset of Y, then $F(Y, A)$:

> for all X and for all A, if $P(X, A)$ then, for all proofs Π of A from any subset Y of X, there is a proof Σ of A from some subset Z of Y such that $F(Z, A)$.

Results like this are called *filters.* They say, essentially, that if $P(X, A)$ then the problem X ?– A has a proof only if $F(X, A)$. So if we are given X ?– A, and can determine that it has property P but lacks property F, then we know that there is no proof to be had.

One has to be careful to prosecute the enquiry into constraints with great care, so as to avoid producing an incomplete proof-finder for one's chosen logical system. One has to pay attention, that is, to the *compossibility* of constraints. For suppose one has a series of P_iF_i-normal form theorems $(i = 1, \ldots, n)$. One may have a provable problem that satisfies all the P_i. If one has constrained one's search by using all of the corresponding F_i, then

one must be assured that the F_i are compossible – that is, that there will indeed be a proof satisfying all the properties F_i.

Think of a completeness-conserving contraint F as a spotlight on a surface whose points are proofs. Compossibility then amounts to this: with several spotlights in play, one wants to be sure that there is a region that they all illuminate. When one's constraints are not thus compossible, then one is forced to choose different combinations from among them that are. And here lies the prospect (for computational logic as a branch of cognitive science) of being more or less faithful to the repertoire of human logical competence. Some completeness-conserving constraints may force proofs into a form that is highly unlikely to be happened upon by human reasoners. Others may turn up proofs on which all human avenues of logical enquiry converge. The aim is to produce highly readable proofs that are rigorous and detailed formalizations of intuitive lines of human reasoning. The more succinct arguments that human reasoners would produce in actual logical or mathematical discourse will be homomorphs of these formal proofs under a very natural projection.

Another advantage for a computational logician in working with systems of natural deduction rather than, say, with Beth tableaux or the resolution method, is that debugging one's proof-finding programs is much easier. Suppose, for example, that one discovers a provable problem that one's program fails to prove. When one examines the trace of the computation one is much better able, in a natural deduction system, to locate and isolate the characteristic errors through failure to construct various subproofs of the would-be proof. Clauses of the program correspond to rules of inference; calls of clauses correspond to attempted applications of these rules of inference. The subproblems generated correspond to the subproofs required for successful application of those rules of inference. Diagnosis and debugging are very easy in such a nested environment.

A decidable logic is the simplest case of a decidable theory based on it. One can implement a decidable theory in a variety of ways, ranging from the evidentially miserly to the evidentially generous. Take the *decision problem* for a theory T:

> Find correct Yes/No answers to problems of the form X ?– A, where X is a finite set of premisses and A is a conclusion, and the question mark concerns the relation of deducibility within T.

There is a minimal response to this problem:

Bare oracles One could give a bare oracle for the decision problem: a program that computed Yes/No answers and gave nothing else as output. This would be a case of extreme evidential miserliness. (It goes without saying that the answers would have to be correct. This is true also of the programs involved in the remaining responses.)

Then there are two intermediate responses:

Proof-finders: One can give full reasons for Yes answers, in the form of proofs. Proofs are finite objects that we can check for correctness. It matters not whether we check the correctness of proofs "by hand" or by means of yet another program – a proof-checker. Proof-checkers are not to be confused with proof-finders. The former verify proofhood. The latter find proofs. If we know that the proof-finding program is correct, however, we will not have to check them. Instead, we can use them to convince others who may be sceptical about positive answers. I call such a program a proof-finder.

A proof-finder for a given theory may be complete or incomplete. A complete proof-finder is one that gives at least one proof for each true statement of deducibility in the theory. An example, by contrast, of an incomplete proof-finder for a theory would be one which implemented an axiomatic system of arithmetic (such as Peano-Dedekind arithmetic), for the theory consisting of all true sentences of arithmetic.

A proof-finder could be *complete* – insofar as it would eventually, for any given provable problem, find a proof of it – and yet fail, on some unprovable problems, to yield even bare negative decisions. On these problems it would not terminate. A complete and *bounded* proof-finder is one which will eventually terminate with a negative verdict on any unprovable problem. Even with a complete and bounded proof-finder one cannot tell, from its failure to respond by any given time, whether it had not yet had time to find a proof, or whether indeed there was no proof to be had. One simply has to wait. All one knows is that one will not have to wait for ever.

A *hypercomplete* proof-finder would do even better than a merely complete one: it would provide a distinct formal but faithful representation, in the form of a proof, for each of the possibly many different informal arguments that might serve to establish the validity of the transition, in the theory, from premisses X to conclusion A. Hypercompleteness is of necessity an informal notion, like that of computability; but it is important to bear in mind as an ultimate desideratum. It will turn out, however, that if complete proof-finders written in any version of Prolog based on a depth-first strategy are to have remotely tractable tasks, they must abjure hypercompleteness. Otherwise the proliferation of alternative proofs on backtracking will greatly delay the making of correct negative (and positive) decisions.

Best of all proof-finders would be a hypercomplete and bounded one, which was able to arrange all alternative proofs in some order of ascending complexity. The notion of such order, however, has yet to receive a satisfactory theoretical analysis.

Counterexample-finders: One can give full reasons for No answers, in the form of counterexamples. Finite counterexamples, like proofs, may be checked for correctness. Counterexamples may be used to convince sceptics about negative answers (but see below).

In the case of first-order mathematical reasoning, a counterexample will be a structure – finite or infinite – that forces all the premisses (makes

them true), but does not force the conclusion. There are interesting philosophical problems, however, concerning the status of counterexamples to X ?– A as supposedly semantic objects distinct from proofs. One can have philosophical reservations about the epistemic force or persuasive power of an infinite counterexample *qua* semantic object. In the case of an infinite counterexample, it could be maintained that what we really have in mind is a well-known theory, given by a set of axioms widely assumed to be consistent, and enjoying the counterexample as a model, and that the counterexemplification of X ?– A consists in the proof-theoretic facts that there are proofs of each of the premisses in X from those axioms, and a proof that the conclusion A is inconsistent with those axioms. And one can easily contend that a finite counterexample is, once again, simply another way of coding proof-theoretic facts: facts concerning the existence of proofs of each of the premisses in X from (an obviously consistent set of) axioms categorically describing the finite counterexample, and the existence of a proof that the conclusion A is inconsistent with those axioms.[2]

In the propositional case, another kind of counterexample may be *logical matrices*. These provide functional interpretations of the connectives over a set of designated and undesignated values. They have to be sound for the theory in the sense that every true statement of deducibility in that theory preserves designated value from premisses to conclusion under every assignment of values to the propositional variables involved. A matrix counterexample to X ?– A is then an assignment of values to the propositional variables involved in X and in A on which each premiss in X takes a designated value and A takes an undesignated value. This is a generalization of the familiar matrix $\{T, F\}$ for classical propositional logic, with T designated and F undesignated, and the usual truth-functional interpretations for the connectives. A class of sound matrices is complete for the theory if every false statement of deducibility in that theory has a counterexample using a matrix in that class. A sound and complete class of matrices is said to be characteristic for the theory. Another example of a characteristic class of matrices for a theory is the class of Jaskowski matrices for intuitionistic propositional logic. One well-known way of showing that an axiomatisable propositional theory (including a logic) is decidable is to show that it has a characteristic class of finite matrices. For then the decision procedure can exploit two enumerations: one of proofs, by virtue of axiomatisability, and one of the finite sound matrices, testing the latter effectively to see whether they counterexemplify the problem in hand. Eventually either the first enumeration hits on a proof of the problem, or the second enumeration hits on a counterexemplifying finite sound matrix.

Like a proof-finder, a counterexample-finder for a given theory may be complete or incomplete. A complete counterexample-finder is one that gives at least one counterexample for each false statement of deducibility in the

[2]For a fuller develoment of this view, see Tennant 1986.

theory. Even a sound but incomplete class of matrices can be useful for an algorithm of this third kind. But such a class is usually exploited for the fortuitous benefits it might bestow in attempts by a proof-finder to prune the search tree in the space of possible proofs. Its members could be used to counterexemplify sequents generated by inductive breakdown of deductive problems in those logics. This can yield speed-up in proof search in the context of the proof-finder of which the counterexample-finder is a module even though the matrices be drawn from a class that is incomplete for the logic concerned.

It is possible for a complete and bounded proof-finder also to be a counterexample-finder (and of finite ones at that) without much further ado. The trace of a terminated and unsuccessful search by such a proof-finder for a proof in response to X ?– A is, after all, a finitary object that bears effective scrutiny. It can play the epistemic rôle of a finite counterexample to X ?– A to one who knows how to read it.

Finally, we have the full-blown response to the problem, which is to provide programs that I call:

Judicial reasoners: These are both proof-finders and counterexample-finders. That is, they give full *rationes decidendi* for their positive or negative verdicts on deductive problems. The best among these would be the *rationauts*. A rationaut would be able to give the shortest or simplest proof possible as its answer to any provable problem, and be hypercompletely ready to give, on request, all alternative proofs, in normal form, in ascending order of length, complexity, roundaboutness or what one will; and would be able to give the shortest or simplest counterexample possible as its answer to any unprovable problem.

The way I set about the task at hand is to aim, modestly, for a complete and bounded proof-finder for a well-chosen decidable propositional logic. It is not hypercomplete. But its terminated unsuccessful searches can deliver traces playing the role of counterexamples to the unprovable problems concerned. The algorithm for finding proofs is written in such a way, however, as to allow one to insert various filters on the sub-problems generated during the search. We have such filters anyway in connection with past recorded successes and or failures. In just the same way we could incorporate filters exploiting matrices, say, if we so wished. But we do not actually do so. The only filters we employ, apart from those recalling past successes and failures, concern the various sorts of syntactic relationships that subformulae can bear to containing formulae. That is, we try to exploit only the sort of syntactic evidence that it is reasonable to suppose the human reasoner can easily (and perhaps often subconsciously) detect.

One scientific theory can possess the pragmatic virtues of elegance and simplicity to a greater or lesser degree than another. It is a matter of trained taste on the part of practising scientists to decide between them on the basis of such considerations. So too in cognitive science: one can

prefer computational models of human logical competence that exploit only what meets the eye syntactically, so to speak, to models that employ, say, nine-element Heyting algebras in their filtrations during search.

Our search heuristics will encode the effect of generally available transformations on proofs. It is desirable to get as many of these as possible to be invariant across choice of logical system, so that the proof-finder is more easily adaptable to whatever system one chooses to work in.

A first major challenge will be to combine aspects of the *bottom-up* and *top-down* kinds of search that human logicians undertake when trying to construct suasive arguments.

A second major challenge is to program a proof-finder that is short enough to admit of an *informal proof of correctness*. On our approach we justify the inclusion of every clause in the Prolog program that embodies the proof-finder, by appeal to proof-theoretic considerations concerning the logic in question.

A third major challenge is to exploit the notion of the relevance of premisses to a conclusion in a manner that I would call it *endogeneous* to the proof-finder. One does not want the proof-finder to find proofs indiscriminately, and only thereafter produce one that exhibits genuine relevance in all its inference steps. One wants rather to use the requirements of relevance to avoid the irrelevant inferential directions and focus on the relevant.

References

N. Tennant. *Autologic*, Edinbugh: Edinburgh University Press 1992.

N. Tennant. "The Withering Away of Formal Semantics?", *Mind and Language* 1 (1986), 302 318.