A Hybrid System to apply Natural Language Inference over Dependency Trees

Ali Almiman
University of Manchester
UK, Manchester
M13 9PL
a.almiman@mu.edu.sa

Allan Ramsay
University of Manchester
UK, Manchester
M13 9PL
allan.ramsay@cs.man.ac.uk

1 Introduction

In the last decade, there has been a surge of interest in the problem of textual inference, which systems can use to automatically determine whether a hypothesis, H, can be inferred from a given text, T. A variety of approaches have been explored ranging from shallow-but-robust to deep-but-brittle. Systems that have tried to avoid semantic representations have applied shallow techniques on natural language snippets, such as measuring lexical overlaps (Jijkoun and Rijke 2005), extracting pattern-based relations (Romano et al. 2006), or applying approximate matching to predicate-argument structure (Hickl et al. 2006). Although these methods are robust and effective, they are not suitable for problems that require multi-steps of inference, such as in the FraCaS multi-premise problems that follow, because the task that they are designed to address simply does not involve applying sequences of rules:

P1: Both leading tenors are excellent.
P2: Leading tenors who are excellent are indispensable.

H: Both leading tenors are indispensable.

Multi-step inferences are more usually carried out by transforming the given sentences into formal representations and then applying a theorem prover to check inferential validity, as described by Akhmatova (2005). Although these approaches can solve multi-step inferences, they are limited by the difficulty of extracting formal paraphrases from freely occurring texts (MacCartney and Manning 2007). In particular, such systems generally require the input texts to be analysed in terms of a grammar with semantically annotated rules. The widespread occurrence of ambiguous and, worse, extra- and a-grammatical sentences makes this extremely challenging.

In this work, we explore a different way to deal with this problem by developing a theorem prover that accepts natural language dependency trees instead of translating them into a logical format. All the sentences are parsed using a version of MALTParser[1] trained on the Penn TreeBank. This has the advantage that we do not need to annotate the rules of the grammar with semantic interpretations (indeed, there **is** no grammar to annotate) and that we can do at least something useful with ill-formed or otherwise anomalous texts.

2 The Hybrid System

2.1 Building a Theorem Prover

To make an inference engine that can benefit from background knowledge, we must utilize an idea that is similar to existing theorem provers. Since all of the inference rules that we use in this experiment are Horn clauses, we mimicked the strategy that works well for proving Horn clauses in existing theorem provers. Consider the standard rule of modus ponens for propositional logic:

$$B \rightarrow A, B \vdash A \tag{1}$$

This suffices as the basis for a simple backward-chaining inference engine for Horn clauses, which can be used to capture a substantial portion of the kind of knowledge that underpins natural language.

We will rewrite this rule as (2) to enable generalisation to first-order logic and thence to our system. This version of the rule introduces a step which would lead to substantial inefficiency if it were implemented in a simple-minded way for propositional logic, but which leads very naturally

[1]The acronym stands for the Models and Algorithms for Language Technology Parser, and it is freely available at http://www.maltparser.org

Proceedings of Recent Advances in Natural Language Processing, pages 64–70,
Varna, Bulgaria, Sep 4–6 2017.
https://doi.org/10.26615/978-954-452-049-6_010

to the version used in standard first-order theorem provers.

$$B \rightarrow A', A = A', B \vdash A \quad (2)$$

In first-order logic (FOL), a theorem prover can prove (A) if Rule [3] applies, where A is **unified** rather than identified with a term that leads to B.

$$B \rightarrow A', A \oplus A', B \vdash A \quad (3)$$

From the rules [2 and 3], it can be seen that the main difference is the matching relationship between the items A and A'. We have as a general rule that you can infer A from B and $B \rightarrow A'$ if A and A' are equivalent: in propositional logic, two formule are equivalent if they are identical; and in first-order logic they are equivalent if they are unifiable. But this suggests that we could obtain other kinds of logic by exploring other notions of equivalence, e.g. the kind of matching algorithm used in textual entailment systems ($\approx$) as in Rule [4].

$$B \rightarrow A', A \approx A', B \vdash A \quad (4)$$

Before it looks for inference rules, the inference engine checks whether there is any fact that approximately matches the goal using the rule [5]

$$A', A \approx A' \vdash A \quad (5)$$

.

2.1.1 Approximate matching

It is possible to obtain a variety of logics by changing the matching algorithm: Pulman (1997), for instance, proposes a unification algorithm that allows predicates as well as terms to be treated as variable to allow for a range of higher-order phenomena. The approximate matching relation we propose is defined as follows.

Let T and T' be two dependency trees with heads H, H' and sets of daughters D, D', where D and D' may be empty. Then $T \approx T'$ if

(i) H is a hypernym of H', as proposed byBaroni et al. (2012). We use WordNet as a source for determining hypernym relations between open-class words. For instance, *"I found a cat"* $\approx$ *"I found an animal"*, because *"cat"* $\sqsubset$ *"animal"*. In addition to relationships between WordNet open-class words, we made some hierarchical relationships between generalized quantifiers according to suggestions from the literature on natural logic (MacCartney and Manning 2008; Icard 2012; Vendler 1962; Kayne 2007; Poesio 1994; Barwise and Cooper 1981). The highest level of that hierarchy is shared between the following quantifers: *all* = *every* = *each* = *(the* + *plural)*. Some other relationships are: *most* $\sqsubset$ *many*, *many* $\sqsubset$ *some*, *a few* $\sqsubset$ *some* and *each* $\sqsubset$ *several*. If there are three quantifiers (α, β, σ), where $\alpha = \beta$ and $\alpha \sqsubset \sigma$, then we conclude that the relationship is $\beta \sqsubset \sigma$. Therefore, all of the root-level words have the same subsumption relationship with the other quantifiers. For example, from *"all"* = *"every"*, and *"all"* $\sqsubset$ *"some"*, we conclude that *"every"* $\sqsubset$ *"some"*, *each*$\sqsubset$ *"some"* and so on.

(ii) There is an order-preserving mapping m from D into D' such that

  (a) $D_i \approx m(D_i)$ for every D_i in D

  (b) if D'_j is not in the range of m then D'_j is headed by a modifier (i.e. an adjective or a preposition)

The first part of this allows us to match sentences where the premise contains a word that is a hypernym of some word in the hypothesis.

Consider the pair *T: "He saw a man"* and *H: "He saw a human"*, and assume that the parse trees for these two are `[saw: VB, [he: PN], [man: NN, [a: DT]]]` and `[saw: VB, [he: PN], [human: NN, [a: DT]]]`. These trees will match because as we recurse down through them we find that every subtree in the first is headed by a hypernym of the corresponding subtree in the second.

The second part allows us to skip modifiers; thus, if the subtree in T has an extra modifier, the approximate matching algorithm is allowed to skip that modifier and tries to match the rest. For example, *T: "He saw a fat man"* and *H: "He saw a human "*; when the system tries to find the match between `[man: NN, [fat: JJ]]` and `[human: NN]`, it unifies *"man"* with *"human"*, and skips the adjective.

Also, the system is allowed to skip prepositional modifiers, e.g., *T: "He saw a man in*

the gymnasium" and *H: "He saw a human"*, because the subtree *'in the gymnasium'* is headed by a modifier. This makes it possible to cope with examples that could not be handled if we treated the texts as strings and used a string-edit algorithm.

(iii) Both the word-level and the phrase-level matching algorithms are asymmetric. For instance, in *I saw a man* $\subseteq$ *I saw a human*, if someone saw a man that means the person saw a human; however, *I did not see a man* $\not\subseteq$ *I did not see a human*. To handle this issue, we add polarity marking to lead the direction of matching. By default, every node of a parsed tree gets a positive mark (+), which means that the context is positive and the matching direction works from left to right. There is a list of words that swap the direction of matching if they exist; e.g., *"no, not, doubt"*, if any of these words is present, then the polarity marking is swapped for all its daughters. There is a list of words that change the direction of the matching if they occur; e.g., *"no, not, doubt"*. If any of these words occur, it changes the polarity marking for all the following nodes. For instance *"I doubt that he likes it"* is turned into `[(doubt,+), [(I,-)], [(likes,-), [(that,-)], [(he,-)], [(it,-)]]]`.

It is important to note that this matching algorithm is non-deterministic: a tree may contain several modifiers, and it may be that choosing to skip over one rather than another will have consequences at a later stage of analysis.

2.1.2 Backward chaining

As mentioned above, to do a proof this system requires set of facts and a set of inference rules in addition to the goal sentence and tries to deduce the proof using a backward-chaining algorithm. This means that it begins with the goal sentence and looks for a matching sentence in the set of facts. If there is no fact that matches this goal, the system looks through the rules to find an inference rule that leads to the given text, T. If that rule is found, the system tries to prove its antecedent. As proving the antecedent is not guaranteed, this algorithm is also a non-deterministic algorithm. We thus have a combination of non-deterministic algorithms: it is important to interweave these appropriately – it may that the backward-chaining algorithm fails if we match the daughters of a pair of trees one way but will succeed if we match them differently (e.g. by skipping over different modifiers). Given that these are independent sets of choices, maintaining the two choice stacks is a challenge. We deal with this issue by using continuation programming (Landin 1998), using the standard call stack as the backtrack stack amd treating return from a function call as failure.

2.2 Inference rules

The backtracking algorithm outlined above requires us to have a collection inference rules. For instance, to find the relationship between *T:"He saw a man in the gymnasium"* and *H: "He saw a man in the large room with equipment for exercising the body"*, there is an inference rule that states: '`X is a large room with equipment for exercising the body` $\Rightarrow$ `X is a gymnasium.`' This kind of rule is definitional, and there is no available source that we know of that uses such inference rules in natural language representation. In most cases, the existing inference rules either annotate their information with some special symbols, e.g., (...), or written as paraphrasing sentences; e.g., DIRT (the discovery of inference rules from text). The former status does not suit our needs as we are trying to avoid logical annotations, and the latter looks for symmetrical relations while we are looking for asymmetrical relations. From this point of view, we made our set of definitional inference rules in Section [2.2.1]. To solve multiple hypothesis tasks, we are looking for inference rules that contain more than one item on the left hand side (LHS) and one on the right hand side (RHS). Therefore, a set of syllogistic inference rules were extracted from the FraCaS multiple-premise examples, and they are presented in Section [2.2.2].

2.2.1 Definitional inference rules

One of the richest sources in which to find definitional information are language dictionaries. Therefore, we chose a selection of 5,000 words from The MacMillan Dictionary (TMDC) and converted the definitions from the form (word:definition) into an inference rule, such as `X is a defintion` $\Rightarrow$ `X is a word`, where X is a variable. To make this transformation, we make the word:definition relationship

into a sentence and parse it; for instance, we turn the definition *"aged: very old"* into *"X is aged if X is very old"*. The forms of these sentences vary depending on the category of the word. For nouns and adjectives, we turned all the definitions from word:definition into "`X is word if X is a definition`". We have established two different types of definitions for verbs. A verb is called *"intransitive"* if it does not take an object, such as *"He ran"*; and a verb is called *"transitive"* if it requires an object, such as *"He drives a bus for living."* To make generic rules for verbs, we look for indefinite NPs with empty nouns such as *"something"*, *"someone"*, or *"somebody"*, which we replace with variables. For instance, *"afford: to provide something"* has been turned into "`X afford Y if X provide Y`" and *"run: to move quickly to a place using legs and feet"* into *"X run if X move quickly to a place using legs and feet"*. It is very very common to have one-word definitions of adverbs, such as: *"commonly: usually"*, and as adverbs are used as modifiers in English sentences we turn them into "`X commonly Y if X usually Y`". There are cases where an adverbs definition contains more than one word, such as, *"always: on every occasion"*. These words are turned into "`X always Y if X does Y on every occasion`". At this stage, we have a main-clause followed by an if-clause, and each clause has a shared variable. To transform these sentences into inference rules, we replace the conditional word *"if"* with a right arrow (⇒) and bring the if-clause as an antecedent and the main-clause as a consequence of that rule; e.g., "`X is very old ⇒ X is aged`".

2.2.2 Syllogistic inference rules

To handle multi-step inference tasks, inference rules with multiple antecedents were extracted from the FraCaS test suite. Multiple-premise FraCaS examples were used because they have words that are shared in both the hypothesis H and the premise $P's$; also, they are marked for their inferential validity. The task is to replace any open-class shared word into a variable. For instance, the example in Figure [1] is transformed into the inference rule in Figure [2].

P1: "Both leading tenors are excellent."
P2: "Leading tenors who are excellent are indispensable."

H: "Both leading tenors are indispensable."

Figure 1: A multi-premise FraCaS example

P1: "Both X are Y."
P2: "X who are Y are Z."

H: "Both X are Z."

Figure 2: A form of syllogistic inference rule

After parsing these sentences, the final form of the syllogistic inference rule will be:
`[[are: VX, [?Y: NN, [both: DT], [?X: JJ]], [?Z: JJ]], [?Y: NN, [?X: JJ], [are: VX, [who: WP], [?Z: JJ], [are: VX, [?A: JJ]]]] ⇒ [are: VX, [?Y: NN, [both: DT], [?X: JJ]], [?A: JJ]]`
These rules were extracted from all of the two-premise FraCaS problems that are judged to be entailed.

2.3 Proving process

When the theorem prover receives an input of both a premise and a goal, it starts from the goal and works toward the hypothesis. A proof can be found from the first step, such as when the inference engine successfully found a fact that approximately matches the given goal. If none of the facts match the goal, then the theorem prover looks at the inference rules; is there is any rule with a head that matches the goal? When no rule's head matches the goal, then there is no way to prove this goal. If there is only one rule with a matching head, then the inference engine sets the head as a new goal and tries to prove the premise using the same steps as the first goal. This step can be done repeatedly until the engine finds a way of getting to a block. If more than one rule is suitable the engine tries all of them in turn. If it fails with the first rule, then it tries the second, and so on, until it finds the proofif it exists. Every variable in the rules has a value, and the value for free variables is `'???'`. The theorem prover can bind any value

to free variables, and when the proof is complete an unbind step is taken to free these variables.

Algorithm 1 Pseudo-code for the theorem prover

```
1:  prove ( goal, facts, rules, contn )
2:    if goal = [] :
3:      contn()
4:    if isList(goal) :
5:      prove(hd(goal), facts, rules,
      lambda:  prove(tl(goal), facts,
      rules, contn))
6:    else:
7:      for f in facts:
8:        match(goal, f, contn)
9:        ⊙
10:     for LHS → RHS in rules:
11:       match(goal, RHS, lambda:
      prove(LHS, facts, rules, contn))
12:       ⊙
```

In Algorithm [1], there is an illustration of the steps that the theorem prover takes. We reach the points marked with ⊙ only if the previous step fails; i.e., if using the fact or rule did not lead to a proof. At that point, we automatically move to the next choice. Figure 3 shows the interactions between the search for a useful fact or rule and the approximate matching algorithm: note in particular that there are three places where something gets added to the backtrack stack – after a fact is found, after some way of matching a rule to the current goal is found, and after that rule has been chosen. When a branch of the search space fails, the algorithm will uniformly pick any one of these stored choices to explore.

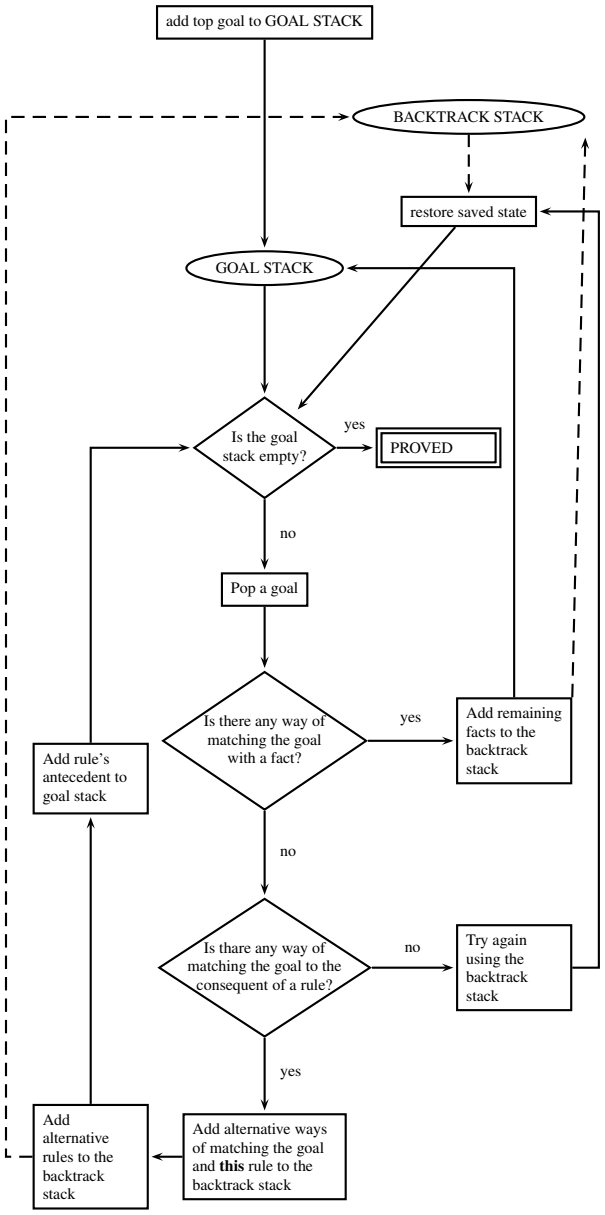

Figure 3: Overall architecture

3 Evaluation and Results

From Section [2.2], we were able to extract 4,613 definitional inference rules. Unsurprisingly, most of these rules were extracted from nouns as they represent 62.9% of the total words, followed by verbs, representing 28.6%. Adjectives comprise only 6.6% of the words, and the lowest number of words are adverbs, comprising only 1.8% of the total number of rules. To evaluate these rules, we have randomly extracted a sample of 100 rules, considering the percentage that each word category covers. Therefore, the greatest number of the sample rules were for nouns, followed by verbs, and so on. This sample was uploaded as an online survey, and we asked native speakers to check whether these are true inference rules or not. For each rule, there are three options. *"Yes"*, if a referee believes the rule is correct; *"No"*, if the rule is wrong; and *"Skip"*, if a referee is not sure about the answer. Each question was limited to four answers, so whenever a question (rule) received four answers it disappeared from the survey to make sure that we have a similar quantity of opinions on all of our sample rules. After obtaining the survey results, we calculated the precision and the recall using the following equations, [6] and [7], respec-

tively:

$$Precision = \frac{R}{100} \quad (6)$$

$$Recall = \frac{R}{100} \times \frac{M}{N} \quad (7)$$

Where N represents the number of definitions that we were looking at, and M denotes the number of definitions that were picked out as being potential rules. The constant (100) denotes the sample domain size, and R denotes the inference rules that were judged as True. From the previous equations, we got 74.0% precision with a recall of 0.68. Using the FraCaS examples, we have applied two different tests: (a) single premise problems test, and (b) a double premise problems test.

For single premise problems, we only employ our approximate matching features as the problems only contain a single premise with a single hypothesis. Amongst the 192 single premise problems that FraCaS test suite has, there are nine problems with an 'undefinite' answer, so they were excluded. MacCartney and Manning (2007) label some answers as 'undefinite' if they lack either hypothesis or well-defined answers. From the remaining 183 problems, we extracted the number of problems for which we could glean the right tree; the number of trees extracted totaled 70 sentences. These problems spread over nine different topics, and most of these topics require more robust parsers and matching techniques, such as anaphora and ellipsis. Therefore, we did the first test on two topics to which we can apply our approximate matching algorithms, which are generalized quantifiers and adjectives. By excluding the non-related topics, the number of problems was reduced to 28. When this system proves a problem, it is marked as a yes; otherwise both no and unknown answers are marked as no. Table [1] details the number of problems that our system observed, and shows how many times the system received a right answer. The four columns under the approximate matching table cell illustrate the status of each technique in each of the four tests. The first test is carried out by switching the word relations on (denoted in the table as: σ), and switching off the other techniques, i.e., detHyps (γ) and skipping modifiers (δ). The last line shows how many right answers can be received when all three techniques are switched on.

Topic	Q	Approximate matching techniques			
		$\sigma = 1$ $\gamma = 0$ $\delta = 0$	$\sigma = 0$ $\gamma = 1$ $\delta = 0$	$\sigma = 0$ $\gamma = 0$ $\delta = 1$	$\sigma = 1$ $\gamma = 1$ $\delta = 1$
Quant	20	12	12	19	19
Adj	8	5	5	6	7
Total	28	17	17	25	26

Table 1: First test on 28 single-premise FraCaS problems

For double-premise problems, we use our set of syllogistic rules that are actually generated from the same set of problems. Since our system is able to bind items to free variables, we were able to make all of the double-premise examples. However, we were looking for generic rules that can be used for more than one example, and consequently we reduced the number of rules used. From the 74 double-premise rules that we have extracted from the set, we were able to find only four generic rules that are able to solve more than one problem; these are shown in Table [2].

Generic rule	Problem No.
G.R(1)	fracas-002
	fracas-003
G.R(2)	fracas-066
	fracas-067
	fracas-068
G.R(3)	fracas-134
	fracas-135
G.R(4)	fracas-157
	fracas-159

Table 2: A list of the FraCaS examples covered by generic rules

The number of generic rules and the examples that they cover could be extended if we did not face the issue of getting different trees for quite similar sentences. This problem, however, is common with data-driven parsers and it has been mentioned by Kouylekov and Magnini (2005), who found more than 30% of similar trees in their test set parsed differently. In FraCaS problems, the shared words are always identical, therefore, there are no available examples to test the approximate matching techniques. Regarding to this problem, we have made up a test by replacing some identical terms to approximatly match terms and checked the sanity of our system on these problems.

4 Conclusion

In this paper, we have explored a new approach to solve natural language inference issues. To make shallow systems less shallow, we applied dependency trees to theorem provers in order to benefit from the advantages of avoiding the step of translating sentences to logic. What we have found is that it is important to have correct trees, and because no reliable, data-driven dependency parser exists, the problem is considered to be very difficult. One way to overcome this problem is by using a chunker instead of parsers, as this system is able to deal with any kind of tree. Another modification is that we add a full stop at the end of each sentence, and then we make that final full stop the root of the tree, and we chunk its daughters. This move makes it easier to control the shapes of trees, but similar to the issue for rule-based parsers that there is no complete set of rules can be written.

References

Elena Akhmatova. 2005. Textual entailment resolution via atomic propositions. In *Proceedings of the PASCAL Challenges Workshop on Recognising Textual Entailment*. Citeseer, volume 150. 1

Marco Baroni, Raffaella Bernardi, Ngoc-Quynh Do, and Chung-chieh Shan. 2012. Entailment above the word level in distributional semantics. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, pages 23–32. 2

Jon Barwise and Robin Cooper. 1981. Generalized quantifiers and natural language. *Linguistics and philosophy* 4(2):159–219. 2

Andrew Hickl, John Williams, Jeremy Bensley, Kirk Roberts, Bryan Rink, and Ying Shi. 2006. Recognizing textual entailment with lccs groundhog system. In *Proceedings of the Second PASCAL Challenges Workshop*. volume 18. 1

Thomas F Icard. 2012. Inclusion and exclusion in natural language. *Studia Logica* pages 1–21. 2

Valentin Jijkoun and M Rijke. 2005. Recognizing textual entailment using lexical similarity. In *Proceedings Pascal 2005 Textual Entailment Challenge Workshop*. 1

Richard S Kayne. 2007. Several, few and many. *Lingua* 117(5):832–858. 2

Milen Kouylekov and Bernardo Magnini. 2005. Recognizing textual entailment with tree edit distance algorithms. In *Proceedings of the First Challenge Workshop Recognising Textual Entailment*. pages 17–20. 6

Peter J Landin. 1998. A generalization of jumps and labels. *Higher Order Symbolic Computation* 11(2):125–143. 3

Bill MacCartney and Christopher D Manning. 2007. Natural logic for textual inference. In *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*. Association for Computational Linguistics, pages 193–200. 1, 6

Bill MacCartney and Christopher D Manning. 2008. Modeling semantic containment and exclusion in natural language inference. In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*. Association for Computational Linguistics, pages 521–528. 2

Massimo Poesio. 1994. Discourse interpretation and the scope of operators. Technical report, DTIC Document. 2

S G Pulman. 1997. Higher order unification and the interpretation of focus. *Linguistics & Philosophy* 20(1):73–115. 2

Lorenza Romano, Milen Kouylekov, Idan Szpektor, Ido Dagan, and Alberto Lavelli. 2006. Investigating a generic paraphrase-based approach for relation extraction. In *EACL*. 1

Zeno Vendler. 1962. Each and every, any and all. *Mind* 71(282):145–160. 2