n	A owes B amount u	———	d_1
o	B owes C amount v	———	d_2
p	$u \geq v$	———	d_3
q	A should-pay C amount v	(split1 n o p)	$d_1 \cup d_2 \cup d_3$
r	A should-pay B amount $(u - v)$	(split2 n o p)	$d_1 \cup d_2 \cup d_3$

Figure 1: A rule for settling debts

Controlling the process of reasoning and deduction in practice depends on careful management of the "sources" of belief (the "premises" of logical statements in the systems) as well as constraining active reasoning to only apply to those statements and rules likely to be relevant to achieve a desired conclusion. These two goals are deeply interlinked by the very essence of reasoning; many rules only become "relevant" in the presence of sufficient statements (or needs) supported by believed premises, while the contradictions encountered in the process of reasoning may provide sufficient evidence to reconsider belief in premises.

This circular dependency may create problems if the "relevance" of a rule is not separated from the *data* dependencies of a particular derived statement (i.e. one which was not directly supported by a premise). The allocation of work in a process of reasoning does not mean that the derived statements have a dependency on the allocation itself. Assuming that the statements supported by a particular premise are invariant, the truth of a derived statement that depends on these base statements is itself invariant; it is either true or it is false, independent of the time or mechanism of its derivation.

How can we properly represent and connect the data dependencies of logical operations carried out in a program separate from the process of allocating effort in their evaluation? Here, we can make use of the formalism proposed by Suppes in [**?**]. Suppes's formalism of logic treats the process of logical deduction as separate from the maintenance of the statements/premises on which the deduced statement depends, but relates the two in such a way that we may describe the process of maintaining dependencies in terms of the deductive operations applied on logical statements.

Take, for example, the case where three people, Alyssa, Ben, and Chris, split a check for lunch. Should Chris owe Ben for a previous meal, and Ben similarly owe Alyssa, they might decide to settle their debts at the time they pay the bill using a rule like that in Figure 1. In these rules (split1 and split2), the amount Chris should pay Ben and Alyssa depends not only on why he owes Ben (dependency d_1) and why Ben owes Alyssa the amount he does (dependency d_2) but also their relative amounts (dependency d_3), so that the statements about Chris's debts depend on their union ($d_1 \cup d_2 \cup d_3$).

Although it is true that Chris should pay Ben and Alyssa appropriately, the practical application of this particular rule is contextual. Even though it's quite useful to calculate how much Chris should pay Ben and Alyssa if they are splitting the check at a restaurant, it's not as useful to apply the same rule if

m	A, B, C in restaurant	———	d_0
n	A owes B amount u	———	d_1
o	B owes C amount v	———	d_2
p	$u \geq v$	———	d_3
q	A should-pay C amount v	(split1 m n o p)	$d_1 \cup d_2 \cup d_3$
r	A should-pay B amount $(u - v)$	(split2 m n o p)	$d_1 \cup d_2 \cup d_3$

Figure 2: A rule for settling debts, only considered in a restaurant

the three are celebrating a birthday party for Chris (After all, it would be rude to ask Chris to pay a debt during his own party!) It is possible to represent this using a variant rule like that displayed in Figure 2.

Obviously, if any of the dependencies d_1, d_2 or d_3 in Figure 1 is no longer believed to be true (perhaps Chris is able to argue that he already paid Ben, in which case d_1 is no longer true), then their conclusion is also necessarily false and Chris would no longer owe either Ben *or* Alyssa regardless of which rule we apply. But once it has been shown that Chris should pay Ben and Alyssa, that fact remains true even if the three leave the restaurant (i.e. there is no dependency on d_0 in Figure 2)! In order to properly trace computation, however, it is still important to note *how* we were able to get to the answer. Thus, the *informants* (i.e. the rule split1 or split2 and the statements which caused the investigation of it, m, n, o and p) is still associated with the conclusions q and r.

Consider another scenario, in which a prosecutor seeks to charge a "deadbeat dad", Joe, for failing to support his child (as his mail has been bouncing and his location is currently unknown). According to the federal deadbeat dad law (Figure **??**), the guilt of such a "deadbeat dad" is completely independent of the way in which the prosecutor is motivated to discover that the father has broken the law.[1] It is useful to separate propositions which motivate an investigation into a deadbeat dad (the fact that Joe's location was unknown) from the evidence which supports Joe's guilt.

This particular scenario demonstrates a necessary component of any such system which separates rules which provide motivation from those which provide conclusions: the "unknownness" of a proposition. Here, the prosecutor's motivation is dependent on the "unknownness" of the proposition "Joe is in the same state as his child". The fact that this proposition is neither known to be true nor known to be false is what motivates the actions of the prosecutor. Should the proposition be known to be true, then the prosecutor has no reason to seek prosecution (as it does not violate the federal law), while if the proposition is known to be false, then there is no reason for the prosecutor to expend effort in establishing Joe's location.

[1]This point ignores issues of process which may appear at first to contradict this claim. But, while the evidence presented by a prosecutor to establish guilt must be procured in accordance with the Fourth Amendment, it is generally not the case that the investigation itself be *motivated* by such process.

Propositions are simply statements which have an associated truth value. A truth value may have one of five values in our system, rather than the two values of traditional Boolean logic. In this regard, it is more closely related to four-valued logics such as that described by .[] To the four values true, false, overdefined (contradictory) and underdefined (unknown), we add a second, nuanced version of the underdefined value representing whether a value has been or can be determined. This second value is thus, effectively, $\not\mathit{unknown}(p)$. Although this "definition by opposition" may have little value when "true", "false" and "overdefined" are more descriptive, the existence of this fifth state is useful in use cases which depend on effective allocation of resources for problem solving, especially when used with facts about knowledge held by other actors. If the truth of a fact is "known" by another system (perhaps one of the prosecutor's co-workers can determine Joe's last known address), it may not be necessary for the system to derive the fact from first principles. Instead, a system may be able to defer evaluation of a statement or "shortcut" this derivation (such as by asking a co-worker for the last known address when it is actually needed).

In order to provide accountability and traceability of logical deductions, propositions make use of truth maintenance systems which provide for the management of incompatible premises in support of a true or false value. In order to properly represent the five values of a proposition's truth, however, the proposition itself is "reified" into five distinct statements, "p is true", "p is false", "p is contradictory", "p is known", and "p is unknown". These individual statements then have their own truth stored in separate truth maintenance systems.

Rule-based reasoning may be constructed on top of these truth maintenance systems by connecting them through what is called a "propagator network." In a propagator network, partial information may "propagate" through a network, causing computation to occur as the partial information is gradually refined or revised. In this way, we may conceive of a reasoner as a propagator network which propagates constraints in a logically consistent manner. For example, a rule which states "p is true AND q is false $\rightarrow$ r is true" may be constructed through the use of a propagator which connects the "true" state of proposition p and the "false" state of proposition q to the true state of proposition r, taking the conjunction of the truth values of the former two states.

Consider an insurance company which seeks to charge rates dependent on the risk factor related to unhealthy eating (e.g. eating hot dogs). Assuming that the company has access to personal information (such as accessing Facebook pages of those individuals seeking to be covered under its policy), how might it use this data to support a decision on the rates that it charges an individual under its policies? Conversely, how can an individual react to these decisions and (perhaps) restrict the visibility of public information so as to evaluate the reasons why certain public information should be private?

An insurance company seeking proof that an individual eats unhealthy food may find proof through a number of ways (Table 1), each of which may be implemented as a separate rule in this system. Each of these rules may be transformed into a system of propositions and the propagators for implication and other logical operations which connect them.

$likes(u, food) \rightarrow eats(u, food)$
$cooks(u, food) \rightarrow eats(u, food)$
$likes(u, restaurant) \wedge serves(restaurant, food) \rightarrow eats(u, food)$
$employed\_at(u, restaurant) \wedge serves(restaurant, food) \rightarrow eats(u, food)$

Table 1: Different rules from which the fact that u eats a particular type of food may be derived

These rule-propagators connect the truth-states of *specific, concrete* propositions. As a result, a mechanism is needed to construct appropriate propagators to connect newly established propositions. This is done by storing references to propositions in a separate database which may be indexed for easy pattern matching of propositions.

In our implementation, the database is implemented crudely as a linked list which is iterated over when pattern matching is requested. As this obviously requires a linear scan over the database whenever a rule is made active, this is not a particularly scalable approach to reasoning. However, application of existing approaches to database indexing may be equally useful and significantly more efficient than the crude implementation used here.

But what about matching against existing rules when a new proposition is added? We may turn our pattern matching into a "trigger" on the database so that pattern matching of the body of any rule (in order to determine when a rule's propagators should be created) is done whenever a new proposition is added to the database.

In our system, this is done by considering the database of propositions to be a partial information structure stored in a cell of its own, so that a propagator which searches for new rules executes whenever the database updates (i.e. whenever a proposition is created). This propagator then scans the database and marks any propositions it finds that match the rule's pattern and constructs the appropriate propagator network. As this approach results in the performance of one linear scan of the database for each pattern for each proposition added, this mechanism is understandably inefficient. A better implementation would instead match only against the partial content added to the database (so as to only effectively scan each proposition once.

The syntactic sugar `(rule ((var antecedent)...) consequent...)` sets up a rule by constructing a pattern-match against a particular specification of a proposition and binding a particular truth value-cell to the variable *var*. For example, the *antecedent* `(accepted (a-prop '(?subject eats unhealthy-food)))` performs a pattern-match to find propositions which match the pattern `(subject eats unhealthy-food)`, where the variable *subject* is denoted by prefixing a question-mark. The accepted cell of any such proposition is then obtained and returned to be bound to the variable, as in a standard Lisp let-form.

These pattern-matchers are chained in the order of the variables (so that the first antecedent must be matched before any subsequent antecedent may be matched), and result in variables bound in earlier antecedents to be used as the

values of variables bound in later antecedents. Since no ordering is placed on the antecedents in a rule, this may cause inefficient pattern-matching; an excessive number of antecedents may be attached to the database as triggers if a particularly "loose" constraint is used as the first antecedent (since it may match a large number of propositions which may not be relevant to the final conclusion). Techniques of query optimization may thus help in order to place stronger constraints before looser ones in the antecedent order without impacting the final results.

Once the final antecedent has been matched, the appropriate consequents will cause conditional propagator connections to be made (so that any premises that are necessary but no longer believed will appropriately "deactivate" the rule and, if so described, its consequents).

The consequent of a rule is executed as the body of a lambda function exactly once for each distinct combination of variable bindings of the unbound variables in the antecedent. Executing such as a lambda function allows for additional evaluation of bound values that are not easily captured in concrete statements stored in our dictionary. For example, the $u \geq v$ requirement in Figure 1 is best resolved in this way (there exist an infinite number of true statements of this form, the reasons for which are fundamentally irrelevant for the task of splitting a bar tab.)