Counterfactuals in Kantian causal networks

term paper: kant, logic, and cognition
lecturers: michiel van lambalgen & ricardo pinosio

Department of Philosophy
University of Amsterdam

Mick de Neeve
9305017/mick@live.nl

August 9, 2019

Assignment *Causal networks can be given a semantics of logic programming. How is counterfactual reasoning done in this framework? How does Pearl do counterfactuals? Look at the concept of counterfactual intervention. Counterfactual intervention reasons forward. Kant needs in certain cases to reason backwards. How should counterfactual intervention be cast in Kantian terms?*

Note *This paper refers to two draft papers: [Pin12] and [vLP15]. These have been bundled and included in the file* `deneeve.klc.drafts.pdf`*, which has been provided together with this paper.*

Abstract A causal network can be viewed as a Kantian object of temporal reasoning. The combination of Pearl's counterfactual intervention on the one hand, and Achourioti and Van Lambalgen's inverse systems formalisation of Kant's logic on the other, has implications for the direction of reasoning. For Pearl, intervention is an action which severs causal links from the past. For Kant however, the cognition of the present presupposes an understanding of past events, and a counterfactual is thus better cast as an observation rather than an action. This paper adapts a causal networks semantics by Pinosio which incorporates this idea. Pinosio's semantics turns out to be equivalent to inverse systems.

1 Introduction

In the *Second Analogy* of the *Doctrine of Elements* in his *Critique of Pure Reason* ([Kan81], §*B232/B233*), Kant puts forward his *Principle of temporal sequence according to the law of causality*:

> *Everything that happens (begins to be) presupposes something which it follows in accordance with a rule.* [*A-edition*]
> *All alterations occur in accordance with the law of the connection of cause and effect.* [*B-edition*]

In other words, upon the observation of some event B it is reasonable to presume the existence of some other event A as its cause. One might thus posit a causal rule $A \longrightarrow B$ that functions as an implication. In *The Bloomberg Logic* ([Kan70], page 230), Kant mentions the modus tollens principle for inferences of reason, effectively stating that if B is denied, then so is A.

Simple as this may seem, it is relevant for the analysis of so-called backtracking counterfactuals, of which Pearl gives an example in *Causality: Models, Reasoning, and Inference* [Pea00], section 7.1.2:

> *Consider a two-man firing squad as depicted in Figure 7.1,*[1] *where A, B, C, D, and U stand for the following propositions:*
>
> U = *court orders the execution;*
> C = *captain gives a signal;*
> A = *rifleman A shoots;*
> B = *rifleman B shoots;*
> D = *prisoner dies.*
>
> *Assume that the court's decision is unknown, that both riflemen are accurate, alert, and law-abiding, and that the prisoner is not likely to die from fright or other extraneous causes.*

So event U causes event C, and C subsequently causes event A as well as B, but either one of the latter two is sufficient to subsequently cause event D.

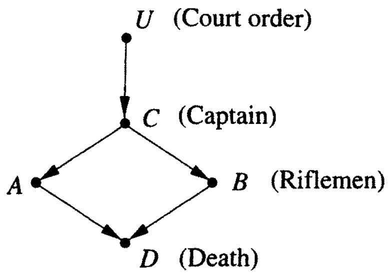

Figure 1: Causal relationships for two-man firing squad (figure 7.1 in [Pea00])

Pearl phrases the following counterfactual: *"If the prisoner is dead, then the prisoner would be dead even if rifleman A had not shot."* The prisoner does not survive in his account because the arrow $C \longrightarrow A$ is severed by performing an intervention that sets the value $A = 0$ (i.e. rifleman A does not shoot). Yet $U = 1$ and hence $C = 1$, i.e. there is a court order and therefore the captain signals, and consequently $B = 1$ and so $D = 1$: rifleman B shoots and the prisoner dies.

Crucially, that $U = 1$ is derived from the actual world. Pearl reasons that because the prisoner dies in actuality, the court order and captain's signal occur in the counterfactual case, too. This is a consequence of how Pearl has intended

[1]Figure 1 in this paper.

his intervention operator (called *do*) to function: it is an instruction to minimally modify the causal network. It essentially (re)sets a value, and from there, reasoning proceeds forward to the eventual conclusion of $D = 1$. Pearl acknowledges the distinction between *doing* and *seeing* ([Pea00], chapter 1, page 23), but in his analysis of counterfactuals he is on the *doing* side of the fence.[2] In this paper, the perspective will instead be *seeing*, i.e. what the consequences are of observing a counterfactual value. This will involve reasoning backwards to causes of observed effects, or making inferences given the absence of causes.

For the present case of the two-man firing squad, in the event where the counterfactual of rifleman A not firing is treated as an observation instead of an action, Kant would plausibly reason as follows. Given the causal rule $C \longrightarrow A$, the value $A = 0$, or rather $\neg A$, means that the effect A of cause C is denied, and consequently the cause itself is denied. Now given the causal rule $C \longrightarrow B$ the implication is that also, $\neg B$. Then because $A \vee B \longrightarrow D$, the final consequence is that $\neg D$, i.e. in case rifleman A had not shot, the prisoner is not dead. So reasoning has first proceeded backwards to the captain not signalling, using the contraposition $\neg A \longrightarrow \neg C$. And from that it in turn follows that rifleman B did not shoot either.

1.1 Partial models and inverse systems

In Pinosio's *Logic Programming and Causal Networks* [Pin12], causal rules can be represented as logic programs. In Prolog,[3], Pearl's network of figure 1 can be rendered as the program $P = \{(\texttt{d:-a;b}), (\texttt{a:-c}), (\texttt{b:-c}), (\texttt{c:-u})\}$. Pinosio has a semantics for logic programs restricted to the acyclic case, i.e. when there is no recursion, which starts with the empty model, and adds heads, or conclusions of rules, given that the bodies, or the rule antecedents, are coherent with the model produced so far. Since for Pearl, in the counterfactual case (`u`) is assumed to have been inherited from actuality, according to his account the initial model would be $M(P)_0 = \{\texttt{u}\}$. Then since $M(P)_0 \models \texttt{u}$, the head of the rule (`c:-u`) may be added, so $M(P)_1 = \{u, c\}$. Next the head of (`b:-c`) can be added because $M(P)_1 \models \texttt{c}$ therefore $M(P)_2 = \{\texttt{u}, \texttt{c}, \texttt{b}\}$. Finally, because $M(P)_2 \models \texttt{(a;b)}$,[4] $M(P)_3 = \{\texttt{u}, \texttt{c}, \texttt{b}, \texttt{d}\}$. In other words, `d` becomes true, as it would in a disjunction, without rifleman A's not shooting having any substantial role in the reasoning process.

The sequence $M(P)_3 \longrightarrow M(P)_2 \longrightarrow M(P)_1 \longrightarrow M(P)_0 = \{\texttt{u}, \texttt{c}, \texttt{b}, \texttt{d}\} \longrightarrow \{\texttt{u}, \texttt{c}, \texttt{b}\} \longrightarrow \{\texttt{u}, \texttt{c}\} \longrightarrow \{\texttt{u}\}$ can be viewed as an inverse system of retracts following Van Lambalgen and Pinosio's *The Logic of Time and the Continuum in Kant's Critical Philosophy* [vLP15]. This is a temporal version of Achourioti and Van Lambalgen's *A Formalization of Kant's Transcendental Logic* [AvL11]. In the latter study, inverse systems are used to model the synthesis of indeterminate objects of empirical intuition (section 7.1.2), resulting in the mental construction of objects in the world from sensory material, such that these can be consciously thought in a unified way. Broadly speaking, the former paper

[2]Except that the inference that the court has ordered the execution follows from *seeing* that the prisoner is dead in the actual world.

[3]https://www.swi-prolog.org/

[4]The semicolon is the disjunction operator, so this means $a \vee b$.

aims to do something similar with time. It models the oncatenation of events such that a larger event encompassing several constituent events can be thought in a coherent unified representation, or consciousness. In other words, inverse systems are used to model Kant's notion of the infinite divisibility of time (cf. section 2.1).

1.2 Transcendental logic

In [Kan81] §*A62/B87*, Kant describes trancendental logic as containing *"the principles without which no object can be thought at all"*, and as *"a logic of truth"*. Therefore, in [AvL11] as well as [vLP15], inverse systems function as semantic structures.[5] The position adopted in this paper is that for the task of answering a counterfactual question, a causal network can be viewed as an object in the Kantian sense, i.e. an object of experience which represents the conditions for experiencing the two-man firing squad case in terms of rules, which in turn represent the various possibilities which may obtain. These rules are inference rules from ordinary logic,[6], but how they interact, i.e. the object of experience itself, is governed by transcendental logic. Most relevantly, the causal principle that *"Everything that happens (begins to be) presupposes something which it follows in accordance with a rule"* (referenced in section 1) is part of transcendental logic: it is an a priori judgment that is presupposed in order for experience to be possible, and for the case at hand it has as a consequence that if rifleman A did not fire, this state of affairs did not come out of the blue. In other words, it is transcendental logic that dictates how to arrive at the counterfactual judgment concerning the prisoner, and in this case it turns out that one must reason backwards.

The aim for the remainder of this paper is to accomodate reasoning backwards in a causal network in terms of inverse systems.

2 Causal networks and Kant's causality

This section will provide some background on Kant's views on causality. As indicated, transcendental logic is about the principles which Kant considered necessary in order for thinking in the first place. Kant's general logic pertains, among other things, to inference types such as modus ponens and modus tollens.

The former is the basic proof mechanism for Prolog: function statements, or rules, have the form `y:-x` and express the logical relation $x \to y$; the idea being to verify x in order to prove y. Modus tollens is not supported in Prolog by default, since it would render the above statement as `not(x):-not(y)`, but `not(y)` does not have to be proved in Prolog because the negation of x is defined as the inability to prove x,[7] and hence $\neg x$ does not need to be explicitly

[5]And since inverse systems implictly feature in [Pin12], they are semantic structures there, too.

[6]Or more precisely Kant's general logic.

[7]This is known as *negation by failure*, and boils down to the so-called closed-world assumption: anything that is not present in, or derivable via the knowledge base, is assumed to be false.

present. However, in order to model reasoning backwards in a causal network, such contraposed rules will need to be explicitly expressed.

According to Kant, the use of reason is dependent on the assumption that things which happen in the world, happen according to universal rules (*Metafysik L*, [Kan75], page 39). An inference from a ground x to a consequence y ($x \rightarrow y$, or `y:-x`), occurs via a logical connection between ground and consequence ([*ibid*], page 164). The same holds for inferences from the denial of consequence to the denial of ground (`not(x):-not(y)`). It is these kinds of rules which are specified in a causal network, and the transcendental connections are between partial models, with which [AvL11] and [vLP15] model the unification of partial objects of experience into a single one. Kant's principle *"Everything that happens (begins to be) presupposes something which it follows in accordance with a rule"* is transcendental, and indicates that the unification of objects of experience should proceed backwards, in the direction of explanation, before it goes forward, in the direction of deduction.

To recap, the above principle is the *Principle of temporal sequence according to the law of causality*. For Kant, this is a universal law that lies at the basis of the human capacity for cognition, and it is presupposed rather than infered from experience. He suggests that resorting to experience to derive the concept of cause, as Hume did [Hum48], boils down to intellectual laziness ([Kan81], §*B124*). Even if experience offers ample examples of the regularity of appearances to arrive at the concept of cause by induction, doing so would not endow it with the required universality, since experience can only lead to the judgement that a rule is possible ([*ibid*], §*A92*) not that it is necessary. What is more, as Kant points out in §*A199/B244*, appearances of preceding times necessarily determine those of subsequent ones, and so arguably, even if causality were infered from experience, the temporal sequences that gave rise to that conjecture would have to presume it anyway.

2.1 Kant and time

For Kant, the concepts of time and causality are inextricably linked. Time, like space, is an a priori intuition, or, as Kant states in [Kan81] §*A32*, a pure form of sensible intuition which is given before any experience occurs. In the same paragraph he points out that different times are parts of the same time, and in §*B48* that time is unlimited. According to Van Lambalgen and Pinosio ([vLP15], page 4) the sense of infinity that Kant refers to here is not in the sense of an outward stretch, but in the inward sense of divisibility. In other words, time is infinitely divisible.

Time can be experienced in three modes: persistence, succession and simultaneity ([Kan81], §*A177/B219*). Succession, but also persistence, is linked to the concept of alteration (of magnitudes), and hence to causality (cf. §*B232/B233*). Broadly speaking, succession or change in what is observed occurs during intervals of time, while persistence happens during instants. An instant in Kant's philosophy of time is not a 'point' of time; it is like an interval except that it has not been analysed. So an instant could in principle be opened up and made into an interval, by further analysing what was initially seen as persistence, and observing change at a higher resolution. Van Lambalgen and Pinosio ([vLP15], page 5) give the example of reading the hands of a clock. During a given interval $[I, J]$ the large hand could be observed as moving from 10 to 15 minutes, but the instant J might be opened into $[J', J'']$ for a more precise analysis, with the hand moving from $14\frac{1}{2}$ to $15\frac{1}{2}$ minutes.

In a similar move from coarseness to precision, the causal network of figure 1 could be viewed as representing a moment, at least for the interval $[C, D]$, i.e. omitting the court order U. The variables C, A, B, and D (i.e. the nodes) are instants, while the arrows (the edges) represent intervals. The network might initially be rendered as consisting only of the interval $[C, D]$, with the rule `d:-c` expressing that the prisoner's death is caused by the captain's signal. In that case C would be an instant which one could, for the sake of obtaining more precision, (re-)divide into C, A, and B, to express that the causality or effect of the captain's signal upon the prisoner's life is mediated through the riflemen A and B.

A model of causal cognition that follows Kant should recognise and encode that in the second case, the understanding of the prisoner's survival or death is linked to the captain's signalling, just as it is in the first case. In other words, both cases should be structurally similar.

3 Causal reasoning and inverse systems

This section is about the formalisation of infinite divisiblity in Van Lambalgen and Pinosio [vLP15], Pinosio's logic programming semantics [Pin12], and how the latter can be adapted to handle backtracking counterfactuals. It also describes an implementation in Prolog.

As indicated in section 1.1, in Achourioti and Van Lambalgen [AvL11], inverse systems are used to model Kant's synthesis of objects of emperical intuition. The idea is that partial objects of experience are – subconsciously – concatenated to create a coherent, conscious experience of the world. In Van Lambalgen and Pinosio [vLP15], inverse systems are used to model Kant's infinite divisibility of time. This paper uses the implicit inverse systems semantics of Pinosio [Pin12], but now so that partial objects are concatenated coherently in order to model the conscious application of learned cognition, i.e. logical rules, to obtain a coherent interpretation of event sequences encoded in causal networks.

In *The Jäsche Logic* [Kan00], in §*60*, Kant indicates that inferences of reason are subject to the objective unity of consciousness. For this paper, this is taken as evidence that Kant considered logical inferences to be bound to the same rules of synthesis as empirical intuition. In other words, concatenating partial models should be as sound for logical inference as it is for emperical intuition. Incidentally, in §*55* he explicitly refers to the universal rule of contraposition, stating that denial of the predicate implies denial of the subject, meaning that the falsity of the consequent implies the falsity of the antecedent (which is relevant to what happens in Pearl's two-man firing squad example).

In [vLP15], page 27, Van Lambalgen and Pinosio point to the use of inverse systems to formalise Kant's synthesis of the unity of apperception in [AvL11], i.e. in the context of the aforementioned empirical intuition, but that this is still valid in a temporal setting. So while the focus of [vLP15] is the modeling of subconscious temporal intuitions about event sequences, the focus here is conscious temporal inferences about such sequences. For instance, if in figure 1 the connection between the captain's signalling and the prisoner's fate can be coherently understood, then so can the connection between the captain's signalling and the prisoner's fate when this is mediated through the riflemen.

For precise specifications of inverse systems, the reader is refered to [AvL11] (page 268) and [vLP15] (page 26). Suffice it to say here that the basic idea – in reverse – is to add consequents of coherent antecedents, via a sequence of mappings. The mappings are to observed magnitudes in the world which are associated with points in time, since it is changes in these observed magnitudes which, for Kant, make time 'tick'. A magnitude could be the rising temperature of a stone caused by the sun shining on it (cf. [vLP15], page 40), but in the present case of the firing squad and related instances, magnitudes are restricted to the classical truth values true and false.

3.1 Logic programming and inverse systems

This section aims to show that Pinosio's logic programming semantics, when restricted to an acyclic causal network case, boils down to an inverse system with retracts. In certain cases multiple inverse systems are required; some examples will make this clear.

Pinosio [Pin12] gives a method to construct a Herbrand interpretation given an acyclic logic program. A Herbrand interpretation I is defined as a pair (T, F) of disjoint sets of atoms, to hold atoms that are true (in T) and false (in F). Then for each atom p, $I \models p$ *iff* $p \in T$.[8]

Pinosio defines an operator Φ_P for a program P, which, given a Herbrand interpretation, outputs another interpretation such that:

$$\begin{aligned} T &= \{q \mid \exists q \leftarrow a_1, \ldots, a_n \in P \mid M \models a_1, \ldots, a_n\} \\ F &= \{q \mid \exists q \leftarrow a_1, \ldots, a_n \in P \mid M \not\models a_1, \ldots, a_n\} \end{aligned} \tag{3.1}$$

Or, in Prolog notation:[9]

$$\begin{aligned} T &= \{\mathtt{q} \mid \exists \mathtt{q} \;\mathtt{:-}\; \mathtt{a}_1, \ldots, \mathtt{a}_n \in P \mid M \models \mathtt{a}_1, \ldots, \mathtt{a}_n\} \\ F &= \{\mathtt{q} \mid \exists \mathtt{q} \;\mathtt{:-}\; \mathtt{a}_1, \ldots, \mathtt{a}_n \in P \mid M \not\models \mathtt{a}_1, \ldots, \mathtt{a}_n\} \end{aligned} \tag{3.2}$$

Here, it is assumed that `q` is in the Herbrand base following [Pin12], page 1: everything that may be derived from a given logic program. In this paper, this will be expanded to include contrapositions in order to facilitate backtracking counterfactuals. Incidentally, here a single set M is used instead, for instance,

[8]This is the two-valued logic case. Pinosio also has a three-valued case where atoms can be left undefined, which is omitted in this paper.

[9]The operator '`:-`' is the reversed implication '$\leftarrow$', to be read as 'in case that', or 'if'.

in case that $I \models p$ and $I \not\models q$, then M is written (in Prolog notation) as $M = \{$`p`$,$`not(q)`$\}$. So M is the specification of a partial model – and inverse systems are mappings that concatenate partial models.

Following definition 21 in Van Lambalgen and Pinosio ([vLP15], page 26), partial models thus obtained form a sequence of retracts, noting that at each iteration, the head (consequent) of a logic programming clause is added (i.e. reversing the order as in an inverse system, it is the retraction which is added).

Figure 2 shows the mappings for the two-man firing squad as given in section 1.1, for the standard logic programming case where one would only reason forward. To recap: considering, as Pearl does, that the court order for the execution has been given, the counterfactual of rifleman A not firing is effectively irrelevant, since B fires, killing the prisoner. This follows from analysing the counterfactual as an intervention, which in itself causes said rifleman not to fire, rather than considering what might have brought about this state of affairs.

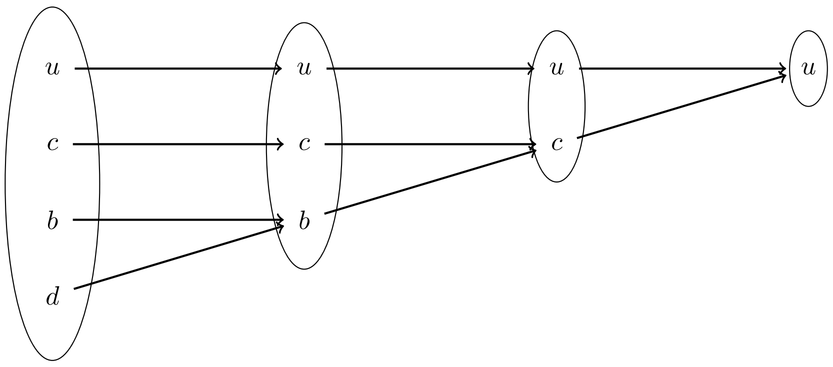

Figure 2: Retraction mappings for Pearl's two-man firing squad, reasoning forward only

Figure 3 on the other hand, depicts the two-man firing squad mappings in case the reasoning process had been like the one described in section 1, i.e. what is conjectured to be akin to Kant's line of thinking according to this paper. Here, the idea is to first heed what might have occured in the past, and only then reason forward given what has been infered contrapositionally.
In both figures, it can be seen which causal rule is used by looking at the value with two incoming arrows. One of these is the identity, the other one is not and indicates the causal rule. For instance, in figure 2 the rightmost value u connects to c, indicating that the rule `c:-u` is used on the basis of the initial model $\{$`u`$\}$. In figure 3, the backwards reasoning step uses `not(c):-not(a)` with initial model $\{$`not(a)`$\}$. I.e. the contraposition of `a:-c` is used.

3.1.1 Prolog implementation of model construction

This section describes an implementation in Prolog which computes a model on the basis of a partial initial model. It also explains the adaptions made to the method in [Pin12] to facilitate backwards reasoning.

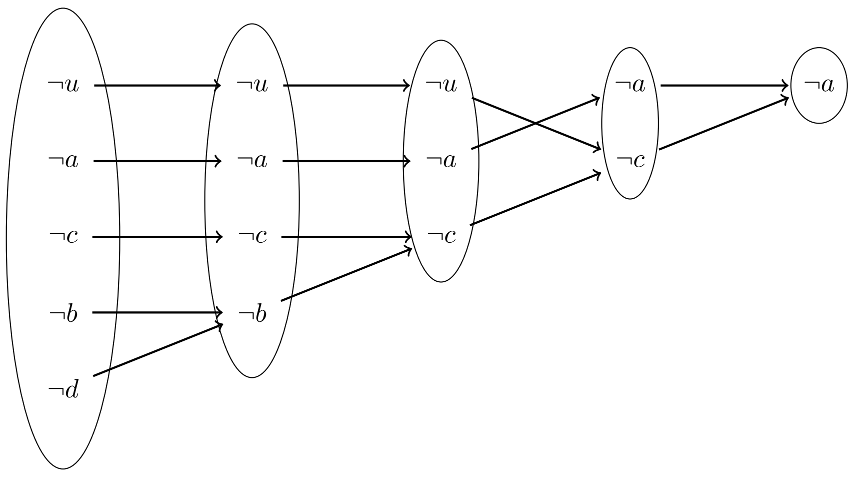

Figure 3: Retraction mappings for Pearl's two-man firing squad, reasoning back first, then forward

The complete Prolog program, called `causalities.pl`, is listed in appendix A at the end of this paper.

3.1.1.1 Core predicates

The predicate `causalities/3` takes as input a causal network expressed as a list of atomic Prolog clauses and list representing the initial model. Actual-world as well as counterfactual values can be put into the initial model, which takes literals of the form `x` or `not(x)`. The model takes the place of the Herbrand interpretations described in [Pin12], which have sets for true as well as for false atoms. Here, truth and falsity are are expressed in a single list, but the idea is the same: the list is a partial model that is filled in a number of iterations. The predicate `causalities/3` then outputs a list of models. The number of models is equal to the number of least fixpoints, in the sense of [Pin12], page 3. More models are generated in the presence of contraposed disjunctions, as described below (cf. `expansions/2`).

The predicate `causality/6` finds the individual models. This predicate terminates when all atoms in the causal network are covered by the model. It broadly follows the method described in [Pin12], page 3, and works by finding a conditional in the causal network clause list that is relevant to the latest value that was added to the current partial model. Since no information concerning the actual world is used, but more importantly to cater for reasoning backwards, the contraposed values of the causal network clauses are appended to the front of the clause list. So for instance in the two-man firing squad problem, with the initial model `[not(a)]`, the contraposition of `a:-c` is used, i.e. `not(c):-not(a)`. Since `not(a)` is true in the initial model, `not(c)` is added. Incidentally, true in a partial model means coherent with it in this case, i.e. not false (cf. [Pin12], definition 5, page 2).

The predicate `expansions/2` takes a causal network clause list including contrapositions, and outputs a list of clause lists with disjunctions in separate lists. For instance, the contraposition of `x:-y,z` is `not(y);not(z):-not(x)`. The predicate transforms `not(y);not(z)` into `not(y):-z` and `not(z):-y`, but then the information that these clauses are the consequent of `not(x)` must be retained, so `not(y);not(z):-not(x)` is ultimately transformed into `(not(y):-z):-not(x)` and `(not(z):-y):-not(x)`. This allows `not(z):-y` to be found and used as a conditional that is relevant to `not(x)` (or to `x`).

The two clause lists that `expansions/2` returns for the causal network `x:-y,z` are `[((not(y):-z):-not(x)), ((not(z):-y):-not(x)), (x:-y,z)]`, and `[((not(z):-y):-not(x)), ((not(y):-z):-not(x)), (x:-y,z)]`. The idea behind these expansions is explained further in section 3.1.1.3.

3.1.1.2 Reasoning backwards

Equation 3.1, following definition 7 in [Pin12] (page 3) indicates that only the consequent `q` of a rule `q :- a` is to be included in a partial model at any iteration.[10] Here, the idea is that if `a` is known to be false, then one would wish to add `not(a)` to the model explicitly. Kant's *Principle of temporal sequence according to the law of causality* referenced at the start of this paper was deemed transcendental in section 2, which is taken to indicate that causal reasoning should proceed backwards before it goes forward.

To accomodate this, equation 3.1 can be adapted, using capital letters instead which may stand for complex formulas as well as a single model set M. Then given a logic programming rule $Q \leftarrow A$ and its contraposition $\neg A \leftarrow \neg Q$, the (partial) model M is augmented as follows:

$$M = \left\{ \begin{array}{rll} A \mid & \exists \neg A \leftarrow \neg Q \in P & \mid M \models Q \\ \neg A \mid & \exists \neg A \leftarrow \neg Q \in P & \mid M \not\models Q \\ & & \\ Q \mid & \exists Q \leftarrow A \in P & \mid M \models A \\ \neg Q \mid & \exists Q \leftarrow A \in P & \mid M \not\models A \end{array} \right\} \tag{3.3}$$

This formulation gives precedence to the contraposition, if Q or $\neg Q$ is present in the model constructed so far. In `causalities.pl` this precedence is handled by adding any infered consequent to the front of the model constructed so far, and having the built-in `member/2` predicate pick out the leftmost relevant rule from the causal network's clause list. This happens within the predicate `value/2`, which makes use of the fact that contraposed causal rules are appended to the front of the clause list.

So for instance in case of the two-man firing squad, the contraposed rules `not(c):-not(a)` and `not(c):-not(b)` appear before `a:-c` and `b:-c`. Given the evidence in the initial model `[not(a)]`, this allows the rule `not(c):-not(a)` to be found and `not(c)` to be infered, leading to the augmented partial model

[10] As already mentioned in footnote 2, the assumption in logic programming is that what is not provably true is considered false (the closed-world assumption), and means that if `a` does not appear as the consequent (or head) of any rule, and is not given as a fact either, one supposes it to be false. This assumption does not hold up here.

`[not(c),not(a)]`.[11] In other words, the rule that rifleman B fires if the captain signals is blocked – precisely because it is first infered that the captain does not signal.

3.1.1.3 Contraposed disjunctions

In Pearl's two-man firing squad, either rifleman's shot suffices for the prisoner to die, so the relation of both to the outcome is disjunctive. In case one were to reason backwards from the supposition that the prisoner survived, i.e. from `not(d)`, the contraposition would be `(not(a),not(b)):-not(d)`, or $\neg d \rightarrow \neg a \wedge \neg b$. In this case the consequence would be certain, but it would not be if both A and B are required for the outcome D, for instance in a more modern example of capital punishment by intravenous injection with two substances A and B. Then, in case all one knows is $\neg d$, reasoning backwards leads to the inference that $\neg a \vee \neg b$. Such situations gave rise to implementing the predicate `expansions/2` as described in section 3.1.1.1.

Knowing that $\neg a \vee \neg b$ leads to two different rulesets, since $\neg a \vee \neg b$ is equivalent to $a \rightarrow \neg b$ but also to $b \rightarrow \neg a$. Using definition 3 in [Pin12] with an empty initial model $M_0 = \emptyset$, one might then first construct $M_1 = \{\neg b\}$ since $M_0 \models a$, and subsequently $M_2 = \{a, \neg b\}$ because $M_1 \not\models b$. This is when one first uses $a \rightarrow \neg b$, and then $b \rightarrow \neg a$. However, when the order of rule application is reversed, $M_1 = \{\neg a\}$ since $M_0 \models b$, and $M_2 = \{\neg a, b\}$ because $M_1 \not\models a$. So depending on the order of application there are two possible models, each of which has a (different) least fixpoint in the sense of [Pin12], page 3.

This is why `expansions/2` splits such disjunctions into ordered sets (as Prolog lists), obtaining in effect $[a \rightarrow \neg b, b \rightarrow \neg a, \neg d]$, and $[b \rightarrow \neg a, a \rightarrow \neg b, \neg d]$ for the above example.

3.2 Counterfactuals in causal networks

This section revisits Pearl's two-man firing squad against the background of inverse systems and counterfactuals, and provides some extra examples of disjunctive counterfactual reasoning.

As mentioned in section 1, Pearl's counterfactual for the two-man firing squad is as follows: *"If the prisoner is dead, then the prisoner would be dead even if rifleman A had not shot"* ([Pea00], section 7.1.2, page 208). Note how the (first) antecedent *"If the prisoner is dead"* sets up the counterfactual world to import everything from the actual world, except that rifleman A did not shoot. This is what Pearl effectively says in [*ibid*], page 210: the inference that the court has given the order for the execution follows from the observation that the prisoner is dead. Consequently, Pearl infers that since rifleman B fired, the prisoner is dead in the counterfactual world, too – and hence that the counterfactual is true.

[11]For completeness, note that it would be in principle be consistent with Pinosio's formulation on page 3 of [Pin12] that using `d:-a` immediately given `[not(a)]`, leads to the addition of `not(d)` to the model, because following definition 3 it holds that `[not(a)]` $\not\models$ `a`, whereby `not(d)` would be infered. However, despite this being the desired conclusion in this case, it would not reflect the desired reasoning process as depicted in figure 3 of this paper.

So Pearl's intervention operator *do*, which severs the incoming causal arrow from C to A in figure 1, can arguably be said to be the cause as to why rifleman A does not fire. Here however, the idea is to instead compute the state of affairs in the counterfactual world, including the cause of rifleman A's not firing, and the subsequent fate of the prisoner.

Intervention in Kantian terms then boils down to nothing more than to set an initial model, in this case $M_0 = \{\neg a\}$, and leaving the causal rules as they have been specified, intact. So the counterfactual would be in this case be phrased as *"If rifleman A had not shot, then the prisoner would be dead"*, and given the final model $M = \{\neg u, \neg a, \neg c, \neg b, \neg d\}$ (cf. figure 3), it evaluates to false.

Taking lowercase letters to be values as above, and uppercase letters as instants (cf. section 2.1), the situation in figure 1 can be viewed as an event comprised of smaller events, and modeling the reasoning process according to figure 3 as a model of the cognition of the various intervals the event comprises, given by the causal ruleset $\{U \longrightarrow C, C \longrightarrow A, C \longrightarrow B, A \vee B \longrightarrow D\}$. From figure 3 it can be read that first, the interval $[C, A]$ is cognised, followed by the intervals $[U, C]$, $[C, B]$, and $[B, D]$. By contrast, in figure 2 the interval $[U, C]$ is cognised first, followed by $[C, B]$ and $[B, D]$. In both cases, the cognitive model ultimately comprises the entire interval $[U, D]$.

3.2.1 Disjunctive counterfactuals

As mentioned in section 3.1.1.3, another capital punishment situation can be envisaged where the prisoner's death results from a conjunction instead of a disjunction: by intravenous injection with two substances A and B, both of which are required for the execution to be succesful. Suppose that administering A on its own results only in (recoverable) sickness, but in case that only B is given, the condemned is left in a permanent vegetative state. For this state of affairs, the following counterfactual can be formulated: *"If the execution was unsuccessful, the prisoner is in a vegetative state"*.

The situation above can be specified using the following causal ruleset $R = \{A \wedge B \longrightarrow D, B \longrightarrow V\}$, with $M_0 = \{\neg d\}$. Or in Prolog using `causalities.pl`, the call would be `causalities([(d:-a,b),(v:-b)], [not(d)])`. This outputs the following:

```
PARTIAL = [not(d)]
PARTIAL = [not(a),not(d)]
PARTIAL = [b,not(a),not(d)]
PARTIAL = [v,b,not(a),not(d)]

PARTIAL = [not(d)]
PARTIAL = [not(b),not(d)]
PARTIAL = [not(v),not(b),not(d)]
PARTIAL = [a,not(v),not(b),not(d)]

MODELS = [[v,b,not(a),not(d)],[a,not(v),not(b),not(d)]]
```

In other words, R given M_0 leads to two models, one in which the prisoner is in a vegetative state, and one in which this is not the case. Figure 4 depicts a split inverse system retraction mappings configuration to reflect this.

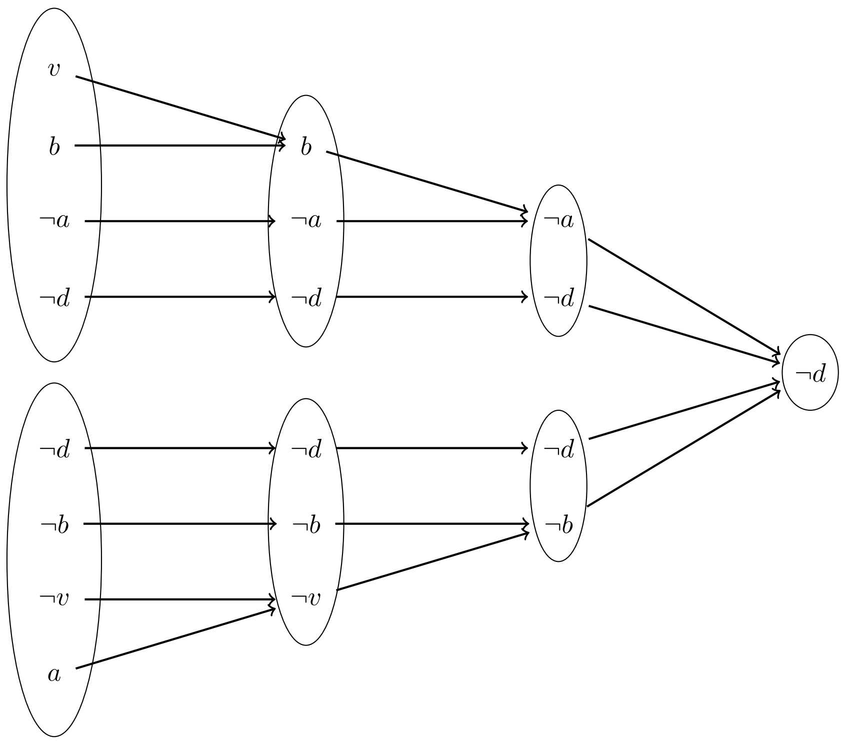

Figure 4: Retraction mappings for contraposed disjunctive backtracking counterfactual

So the counterfactual *"If the execution was unsuccessful, the prisoner is in a vegetative state"* is not simply true or false, but conditional on the disjunctive inference given $\neg d$, which is $\neg a \vee \neg b$. Formally, the answer can be rendered as follows: $\top \leftarrow b \wedge \bot \leftarrow a$. In other words, the counterfactual is true in case that B was administered. Note that since $\neg a \vee \neg b$ is equivalent to $\neg a \leftarrow b$, this also means that A was not administered, which is consistent with the observation that the prisoner is alive.

Finally, also note that this is a counterfactual which can be evaluated just by virtue of reasoning backwards. Cutting the incoming links as resulting from applying Pearl's *do* operator would not allow one to draw any conclusion whatsoever about the value of V.

4 Discussion

In the previous section, a few inverse systems of retracts were drawn, following Van Lambalgen and Pinosio [vLP15]. As indicated in section 1.1, inverse systems are used in [vLP15] to model Kant's notion of the infinite divisibility of time. Section 3.2 describes how temporal cognition is about cognising constituent intervals of a larger interval coherently, so that the larger interval itself can be cognised.

That inverse system are systems of retracts is, according to [vLP15], a consequence of the covering axiom (cf. [*ibid*], page 10, axiom 11). As the paper puts it on p.29, *"for any splitting* $< a, b, d >$ *of* c *there must be an event covering* a, b, d*; one might then just as well retain* c*"* ([*ibid*], page 29).[12] Here, it has been assumed that splitting an event is to analyse it, i.e. c after splitting is no longer an instant, so one could arguably set it as an interval (with a beginning and an end) such that $c = [a, d]$. Then to *"retain* c*"* is in fact to retain $[a, d]$ as a retract of $[[a, b], [b, d]]$. In other words, what is retracted is b, or the end of the interval $[a, b]$ and the beginning of $[b, d]$, i.e. the event which glues time together when it is divided. In terms of causality, having b in both intervals represents the *"continuous action of causality"* ([Kan81], §*B254*).

As Van Lambalgen and Pinosio indicate on page 40 of [vLP15], a causal rule (viz. Kant's *"according to a rule"*, [Kan81], §*A90* and §*A446/B474*) is a relation if it describes data points which might be observed but in fact were not, are also covered by the rule. Data points refer to so-called observations of magnitude, which in the two-man firing squad problem case are truth values. In this paper, unobserved data points have been described implicitly via contraposition, and hence causal rules have effectively been obtained by modus tollens.

5 Concluding remarks

This paper has demonstrated how a causal network makes Kantian causal reasoning rules explicit, in particular how the cognition of temporal sequencing is experienced and cognised according to (intuitive) rules.

An opposition has been made between Kantian counterfactual analysis and that of Pearl, showing how the former is well suited to understanding causal reasoning from observed phenomena to conjectured explanations, by reasoning back to these before reasoning forward to consequents.

Additionally, an example has been given where reasoning backwards is essential to drawing any counterfactual conclusion at all, in particular when the counterfactual value involved introduces disjunctive uncertainty.

References

[AN97] K. Ameriks and S. Naragon, editors. *Lectures on Metaphysics*. The Cambridge Edition of the Works of Immanuel Kant. Cambridge University Press, 1997.

[AvL11] T. Achourioti and M. van Lambalgen. A Formalization of Kant's Transcendental Logic. *The Review of Symbolic Logic*, 4(2):254–289, 2011.

[12]The conventioned of describing temporal intervals with uppercase letters as in section 3.2 is dropped here for the moment.

[Hum48] D. Hume. *An Enquiry concerning Human Understanding.* London: A. Millar, 1748.

[Kan70] I. Kant. The Blomberg Logic, 1770. In Young [You92].

[Kan75] I. Kant. Metaphysik L, mid-1770s. In Ameriks and Naragon [AN97].

[Kan81] I. Kant. *Kritik der reinen Vernunft.* Riga: Hartknoch, 1781. Revised edition, 1789. English translation of both editions: *Critique of Pure Reason*, P. Guyer and A. Wood, Cambridge University Press, 1998.

[Kan00] I. Kant. The Jäsche Logic, 1800. In Young [You92].

[Pea00] J. Pearl. *Causality: Models, Reasoning, and Inference.* Cambridge University Press, 2000. Second edition published in 2009.

[Pin12] R. Pinosio. Logic Programming and Causal Networks. Draft technical report, March 2012.

[vLP15] M. van Lambalgen and R. Pinosio. The Logic of Time and the Continuum in Kant's Critical Philosophy. Draft technical report, University of Amsterdam, February 2015.

[You92] M. Young, editor. *Lectures on Logic.* The Cambridge Edition of the Works of Immanuel Kant. Cambridge University Press, 1992.

A Prolog code

```
/* FILE:        causalities.pl
   REQUIRES:    SWI-Prolog <http://www.swi-prolog.org>
   PURPOSE:     Do counterfactual analysis on acyclic Prolog programs,
                interpreted as Kantian causal networks
   INSTITUTE:   University of Amsterdam
   AUTHOR:      Mick de Neeve <mick@live.nl>
   DATE:        July 31, 2019
*/

/* causalities(+Program, +Initial)
      Finds and writes a set of models for an acyclic Prolog program plus
      initial model, considering in particular the program contraposition
      such that counterfactual models can be handled, where the presence of
      disjunctive contrapositions can give rise to multiple possible models.
*/
causalities(Program, Initial) :-
   nl, causalities(Program, Initial, Models),
   write('MODELS␣=␣'), write(Models).
causalities(Program, Initial, Models) :-
   atoms(Program, Atoms),
   separate(Program, Separated),
   contrapose(Separated, Contraposed),
   append(Contraposed, Separated, Clauses),
   expansions(Clauses, Expansions),
   causalities(Expansions, Atoms, [], _Blocked, Initial, DModels), !,
   equivalences(DModels, Models).
causalities([], _, _, [], _, []).
causalities([Clauses|CTail], Atoms, Block, [Blocked|BTail], Initial, [Model|MTai
   causality(Clauses, Atoms, Block, Blocked, Initial, Model),
   causalities(CTail, Atoms, Block, BTail, Initial, MTail).

/* causality(+Clauses, +Atoms, +Initial, -Model)
      Finds a model for an ordered clause list, with a particular order for
      disjunctions represented as multiple implications, such that the value
      for the first one found clamps those of the others.
*/
causality(_, Atoms, _, _, Model, Modeled) :-
   write('PARTIAL␣=␣'),write(Model), nl,
   covered(Model, Atoms), !,
   modeled(Model, Atoms, Modeled), nl.
causality(Clauses, Atoms, Block, Blocked, Partial, Model) :-
   (  conditional(Clauses, Block, Partial, Conditional),
      New = Block
      ;
      associative(Clauses, Block, New, Partial, Conditional)  ),
   infer(Conditional, Partial, Inferences),
   append(Inferences, Partial, Next),
   causality(Clauses, Atoms, New, Blocked, Next, Model)
   ;
   causality(Clauses, Atoms, Block, Blocked, [?|Partial], Model).
```

```
/* covered(+Model, +Atoms)
      Checks if the model so far covers program atoms (or had to terminate).
*/
covered(_, []).
covered([?|_Model], _Atoms).
covered(Model, [Atom|Tail]) :-
   negate(Atom, Negated),
   (  member(Atom, Model)
      ;
      member(Negated, Model)  ),
   covered(Model, Tail), !.

/* uncovered(+Model, +Atoms, -Uncovered)
      Gets the program atoms not covered by the model.
*/
uncovered(_, [], []).
uncovered(Model, [Atom|ATail], [Atom|UTail]) :-
   not(covered(Model, [Atom])),
   uncovered(Model, ATail, UTail), !.
uncovered(Model, [_|ATail], Uncovered) :-
   uncovered(Model, ATail, Uncovered), !.

/* modeled(+PartialModel, +Atoms, -Modeled)
      Appends uncovered program atoms to a model, after tagging these with a
      question mark, returning the model as-is in case all atoms are covered.
*/
modeled([?|Model], Atoms, Modeled) :-
   uncovered(Model, Atoms, Uncovered),
   unmodeled(Uncovered, Unmodeled),
   append(Unmodeled, Model, Modeled).
modeled(Model, _, Model).

/* unmodeled(+UncoveredAtoms, -TaggedAtoms)
      Tags uncovered program atoms with a question mark.
*/
unmodeled([], []).
unmodeled([Elt|Tail], [?(Elt)|UTail]) :-
   unmodeled(Tail, UTail).

/* conditional(+Clauses, +UnusableClauses, +Model, -Conditional)
      Retrieves an implication relevant to the latest model value found.
*/
conditional(Clauses, Block, Model, (Head:-Body)) :-
   cumulatives(Model, Cumulatives),
   value(Clauses, Model, Cumulatives, Value),
   member((Expression:-Value), Clauses),
   not(member((Expression:-Value), Block)),
   (  conditional(Expression, (Head:-Body))
      ;
      (Head:-Body) = (Expression:-Value)  ),
   exterior(Model, Head, Ext),
   Ext \== [].
conditional(((Head:-Body):-_Tag), (Head:-Body)).
conditional((Head:-Body), (Head:-Body)) :-
   Head \== (_X:-_Y).

/* infer(+Clauses, +Model, -Inferences)
      Checks whether clause bodies are coherent with a (partial) model, and
      if so, infer their heads are true, or else that these are false.
*/
infer([], _, []).
infer([(Head:-Body)|Tail], Model, [Infered|ITail]) :-
   infer(Head, Body, Model, Infered),
   infer(Tail, Model, ITail).
infer((Head:-Body), Model, [Infered]) :-
   infer(Head, Body, Model, Infered).
infer(Head, Body, Model, Infered) :-
   (  coherent(Body, Model),
      Infered = Head
      ;
      negate(Head, Infered)   ), !.

/* associative(+Clauses, +ClausesUsed, +NextUsed, +Model, -Associative)
      Alternative version of conditional, allows extra atoms to be added to
      the partial model temporarily to find conjunctive clause bodies; in
      case a literal is in a contraposed (tagged) disjunction, it is added to
      the model so the original conjunction may be retrieved, but any such
      clause may then not be used by conditional itself.
*/
associative(Clauses, Block, Next, Model, (Associative:-Value)) :-
   [Front|_] = Model,
   atoms([Front], [FAtom]),
   negate(FAtom, NAtom),
   (  member(((Head:-FAtom):-Tag), Clauses),
      Used = ((Head:-FAtom):-Tag)
      ;
      member(((Head:-NAtom):-Tag), Clauses),
      Used = ((Head:-NAtom):-Tag)   ),
   cumulatives([Head,Front|Model], Cumulatives),
   value(Clauses, Model, Cumulatives, Value),
   member((Associative:-Value), Clauses),
   exterior(Model, Associative, Ext),
   Ext \== [],
   associated(Clauses, Used, Associated),
   append(Block, Associated, Next).
```

```
/* associated(+Clauses, +Used, -Associated)
      Finds clauses that are also deemed associated given a variable that was
      used, by retrieving similarly tagged clauses with the same atoms.
*/
associated(Clauses, ((X:-Y):-Z), Associated) :-
   atoms([(X:-Y)], XYAtoms),
   findall(Cand,
   (  member(Cand,Clauses),
      ((H:-B):-Z)=Cand,
      atoms([(H:-B)],HBAtoms),
      intersection(XYAtoms,HBAtoms,Int),
      Int\==[]),
      Associated  ).

/* value(+Clauses, +Model, +Cumulatives, -Value)
      Chooses the first model value that is the condition for a consequence
      in a set of clauses, such that the consequence is not already covered
      by the (partial) model.
*/
value(Clauses, Model, Cumulatives, Value) :-
   polarise(Cumulatives, Value),
   member((Head:-Value), Clauses),
   atoms([Head], Atoms),
   not(covered(Atoms, Model)).

/* coherent(+Body, +Model)
      Tests for coherence with a partial model, this is considered true in
      case that what is to be added is not inconsistent, so it depends on
      what is added first.
*/
coherent((Condition,Rest), Model) :-
   coherent(Condition, Model),
   coherent(Rest, Model), !.
coherent((Condition;Rest), Model) :-
   coherent(Condition, Model), !
   ;
   coherent(Rest, Model), !.
coherent(Condition, Model) :-
   simple(Condition),
   negate(Condition, Negated),
   not(member(Negated, Model)), !.

/* exterior(+Model, +Expression, -Exterior)
      Finds atomic parts of an expression that are not covered by a model.
*/
exterior(Model, Expression, Exterior) :-
   Expression \= (_X:-_Y),
   atoms(Model, MAtoms),
   atoms([Expression], EAtoms),
   ord_intersection(MAtoms, EAtoms, Int),
   ord_subtract(EAtoms, Int, Exterior).
```

```
/* negate(+Expression, -Negation)
      Gets the negation of a conditional expression.
*/
negate((X,Y), (NX;NY)) :-
   negate(X, NX),
   negate(Y, NY), !.
negate((X;Y), (NX,NY)) :-
   negate(X, NX),
   negate(Y, NY), !.
negate(not(X), X) :- !.
negate(X, not(X)) :- !.

/* contrapose(+Clause, -Contraposition)
      Gets the contraposition of an implication.
*/
contrapose([], []).
contrapose([H|T], [CH|CT] ):-
   contrapose(H, CH),
   contrapose(T, CT).
contrapose((X:-Y), (NY:-NX)) :-
   negate(X, NX),
   negate(Y, NY), !.

/* simple(+Expression)
      Tests if an expression is either a propositional atom or its negation.
*/
simple(X) :-
   negate(X, NX),
   (  atom(X)
      ;
      atom(NX)   ).

/* absolute(+Expression)
      Gets the absolute value of a propositional atom.
*/
absolute(not(X), X).
absolute(X, X).


/* atoms(+Clauses, -Atoms)
      Extracts atoms from a list of clauses.
*/
atoms([], []).
atoms([[V]], Atoms) :-
   atoms([V], Atoms), !.
atoms([Clause|CTail], Atoms) :-
   Clause =.. [_,Value|Rest],
   atoms([Value], ValueAtoms),
   atoms(Rest, RestAtoms),
   atoms(CTail, TailAtoms),
   append(ValueAtoms, RestAtoms, VRAtoms),
   append(VRAtoms, TailAtoms, ATail),
   sort(ATail, Atoms), !.
```

```
atoms(Value, Atom) :-
   absolute(Value, Atom).

/* separate(+Program, -Separated)
      Finds clauses with disjunctions in the bodies and separates them into
      multiply-headed clauses.
*/
separate([], []).
separate([(H:-(X;Y))|Tail], Separated) :-
   separate([(H:-Y)], Rest),
   separate(Tail, Next),
   append(Rest, Next, STail),
   append([(H:-X)], STail, Separated), !.
separate([H|Tail], [H|STail]) :-
   separate(Tail, STail).

/* expansions(+Clauses, -Expansions)
      Expands clause list into sublists for each disjunctive implication,
      that may be set-equivalent but different in order, which can result in
      different implications being found and clamping different truth values.
*/
expansions(Clauses, Expansions) :-
   expanded(Clauses, Expanded),
   disjoined(Expanded, Disjoined),
   conjoined(Disjoined, Expansions).

/* expanded(+Clauses, -Expanded)
      Expands clause list into disjunctive implication sublists, adding
      their complements (equivalent disjunctions) and remainders
      (non-disjunctive clauses).
*/
expanded(Clauses, Expanded) :-
   expand(Clauses, Expansions, Others),
   (  Expansions == [],
      Expanded = [Others]   % Expanded = Others
      ;
      complements(Expansions, EComplements),
      remainders(EComplements, 0, Expansions, Others, Expanded), !
).

/* expand(+Clauses, -Expansions, -Others)
      Finds all implicative representations for each disjunction, tagging
      them with their condition and separating them from non-disjunctions.
*/
expand([], [], []).
expand([((X;Y):-Z)|T], [F|E], O) :-
   findall(((H:-B):-Z), rulify((X;Y),(H:-B)), F),
   expand(T, E, O).
expand([X|T], E, [X|O]) :-
   expand(T, E, O).
```

```
/* complements(+Expansions, -ExpWithComplements)
      Adds complements to the sets of disjunctive expansions, since such an
      expansion comprises multiple implications, which are all added.
*/
complements([], []).
complements([Expansions|ETail], [ExpansionsWithComplements|ECTail]) :-
   complements(Expansions, Expansions, ExpansionsWithComplements),
   complements(ETail, ECTail).
complements([], _, []).
complements([Expansion|ETail], Expansions, [Expansion/Complement|ECTail]) :-
   delete(Expansions, Expansion, Complement),
   complements(ETail, Expansions, ECTail).

/* remainders(+ExpWithComplements, +N, +Exp, +Complements, -Remainders)
      Adds the remaining clauses that are not part of the sets of
      disjunctions to these sets.
*/
remainders([], _, _, _, []).
remainders([E/C|ECTail], N, Expanded, Others, [E/C/R|ECRTail]) :-
   nth0(N, Expanded, Expansion),
   delete(Expanded, Expansion, Deleted),
   flatten(Deleted, F),
   append(F, Others, R),
   remainders(ECTail, N, Expanded, Others, ECRTail).
remainders([EC|ECTail], N, Expanded, Others, [ECR|ECRTail]) :-
   remainders(EC, N, Expanded, Others, ECR),
  M is N + 1,
   remainders(ECTail, M, Expanded, Others, ECRTail).

/* rulify(+Disjunction, -DisjunctionAsRule)
      Turns a disjunction into a set of implications.
*/
rulify((Disjunct;Rest), (Selected:-Body)) :-
   rulify((Disjunct;Rest), Selected, Remainder),
   negate(Remainder, Body).
rulify(Disjunction, Disjunct, Rest) :-
   listify(Disjunction, Disjuncts), !,
   select(Disjunct, Disjuncts, Remainder),
   simple(Disjunct),
   disjunctify(Remainder, Rest).

/* listify(+Junction, -List)
      Turns conjunction or disjunction into list of conjuncts or disjuncts.
*/
listify((Conjunct,Rest), [Conjunct|Tail]) :-
   listify(Rest, Tail).
listify((Disjunct;Rest), [Disjunct|Tail]) :-
   listify(Rest, Tail).
listify(Junct, [Junct]).
```

```
/*  conjunctify(+ConjunctsList, -Conjunction)
      Turns a list of conjuncts into a conjunction.
*/
conjunctify([Conjunct|Tail], (Conjunct,Rest)) :-
   conjunctify(Tail, Rest).
conjunctify([Conjunct], Conjunct).

/*  disjunctify(+DisjunctsList, -Disjunction)
      Turns a list of disjuncts into a disjunction.
*/
disjunctify([Disjunct|Tail], (Disjunct;Rest)) :-
   disjunctify(Tail, Rest).
disjunctify([Disjunct], Disjunct).

/*  disjoined(+ExpWithComplementsAndRemainders, -ExpAsSublists)
      Combines the complements of disjuncts-as-rules and the remaining
      clauses, retaining disjoined nested representation of a sublist per
      disjunction and a sublist per disjunct.
*/
disjoined([], []).
disjoined([E/C/R|ETail], [Disjoined|DTail]) :-
   append([E], C, Appended),
   append(Appended, R, Disjoined),
   disjoined(ETail, DTail), !.
disjoined([Expansion|ETail], [Disjoined|DTail]) :-
   disjoined(Expansion, Disjoined),
   disjoined(ETail, DTail), !.
disjoined([X], [X]).  %disjoined([X], [[X]]).

/*  conjoined(+ExpansionsAsSublists, -ExpansionsAsLists)
      Conjoins list of sublists for each disjunct separately into a list of
      lists for each disjunct, so that clause lists for each disjunction only
      differ in order (of processing).
*/
conjoined(D, C) :-
   conjoined(D, [], C).
conjoined([], C, C).
conjoined([[H|T]|L], Cur, Conjoined) :-
   not(is_list(H)),
   append(Cur, [[H|T]], Next),
   conjoined(L, Next, Conjoined), !.
conjoined([H|T], Cur, Conjoined) :-
   conjoined(H, Cur, Next),
   conjoined(T, Next, Conjoined), !.

/*  equivalences(+Models, -TrimmedModels)
      Removes equivalent models from a list of models.
*/
equivalences(Models, ModEqs) :-
   equivalences(Models, Models, ModTags),
   rmeq(ModTags, Eqs),
   rmeq(Models, Eqs, ModDup),
   rmeq(ModDup, ModDb),
   rmdb(ModDb, ModEqs).
```

```
equivalences([], _, []).
equivalences([Model|Tail], Models, [Model/Eqs|ETail]) :-
   delete(Models, Model, Complement),
   equivalent(Model, Complement, Eqs), !,
   equivalences(Tail, Models, ETail).

/* equivalent(+Models, +Models, -TaggedModels)
      Tags models with their equivalent models.
*/
equivalent(_, [], []).
equivalent(Model, [Complement|CTail], [Model,Complement|ETail]) :-
   equivalent(Model, Complement), !,
   equivalent(Model, CTail, ETail).
equivalent(Model, [_|CTail], [Model|ETail]) :-
   equivalent(Model, CTail, ETail), !.
equivalent([], _).
equivalent([Value|Tail], Model) :-
   member(Value, Model), !,
   equivalent(Tail, Model).

/* rmeq(+Model, +ModelsWithEquivalences, -RmModels)
      Removes equivalent models given a list of models tagged with these
      equivalences.
*/
rmeq([], _, []).
rmeq([Model|Tail], ModEqs, [Model|DTail]) :-
   member(Model/Eqs, ModEqs),
   subtract(Tail, Eqs, NewTail),
   rmeq(NewTail, ModEqs, DTail), !.
rmeq([], []).
rmeq([Model/EqLs|Tail], [Model/Eq|ETail]) :-
   sort(EqLs, Sorted),
   delete(Sorted, Model, Eq),
   rmeq(Tail, ETail), !.
rmeq([Head|Tail], [Head|ETail]) :-
   delete(Tail, Head, NewTail),
   rmeq(NewTail, ETail).

/* rmdb(+ModDup, -Mods)
      Removes duplicate models ensuing from complex disjunctions.
*/
rmdb([], []).
rmdb([List|Lists], [RmList|RmLists]) :-
   reverse(List, ReversedList),
   rmeq(ReversedList, RmEqReversedList),
   reverse(RmEqReversedList, RmList),
   rmdb(Lists, RmLists).
```

```
/* polarisations(+Cumulatives, -Conjunction)
      Creates polarised possible conjunction from (partial) model subsets.
*/
polarise(Cumulatives, Conjunction) :-
   member(Cumulative, Cumulatives),
   permutation(Cumulative, Permutation),
   conjunctions(Permutation, Conjunctions),
   member(Conjunction, Conjunctions).

/* conjunctions(+Propositions, -Conjunctions)
      Turns a set of atomic propositions into possible conjunctions,
      including negations.
*/
conjunctions(Cumulative, Conjunctions) :-
   length(Cumulative, Len),
   numlist(0, Len, Nums),
   conjunctions(Cumulative, Len, Nums, Conjunctions), !.
conjunctions(_, _, [], []).
conjunctions(Cumulative, Len, [Num|NTail], Conjunctions) :-
   conjuncts(Cumulative, Len, Num, Conjuncts),
   conjunctions(Cumulative, Len, NTail, PTail),
   append(Conjuncts, PTail, Conjunctions).

/* conjuncts(+Propositions, +Length, +Template, -Conjunctions)
      Makes conjunctions out of a set of propositions based on a template
      that indicates which propositions to negate.
*/
conjuncts(Cumulative, Len, Num, Conjunctions) :-
   negations(Len, Num, Flips),
   conjuncts(Cumulative, Flips, Conjunctions).
conjuncts(_, [], []).
conjuncts(Propositions, [Flip|FTail], [CFlipped|CTail]) :-
   flip(Propositions, Flip, Flipped),
   conjunctify(Flipped, CFlipped),
   conjuncts(Propositions, FTail, CTail).

/* flip(+Propositions, +Template, -Flipped)
      Uses template of ones and zeros to flip the values of a list of
      propositions.
*/
flip([], [], []).
flip([Value|VTail], [1|FTail], [Negated|PTail]) :-
   negate(Value, Negated),
   flip(VTail, FTail, PTail).
flip([Value|VTail], [0|FTail], [Value|PTail]) :-
   flip(VTail, FTail, PTail).
```

```
/* negations(+Sizes, +Negations, -List)
      Generates list of sublist templates which indicate (with the value 1)
      the positions in a list of propositional values where negations are to
      be applied.
*/
negations(N, K, Flips) :-
  M is N - K,
   values(1, K, Ones),
   values(0, M, Zeros),
   append(Ones, Zeros, Base),
   findall(Perm, permutation(Base,Perm), Found),
   sort(Found, Sorted),
   reverse(Sorted, Flips).

/* cumulatives(+List, -Cumulatives)
      Creates a list of cumulative sublists.
*/
cumulatives(List, Cumulatives) :-
   findall(Sub, sublist(List,Sub), Res),
   sortlists(Res, Cumulatives).

/* values(+Element, +Length, -List)
      Creates list of given elements and length
*/
values(_, 0, []) :- !.
values(Elt, N, [Elt|Tail]) :-
  M is N - 1,
   values(Elt, M, Tail).

/* sublist(+List, -Sublist)
      Non-deterministic ordered list version of subset.
*/
sublist([], []).
sublist([Head|Tail], [Head|Sub]) :-
   sublist(Tail, Sub).
sublist([_|Tail], Sub) :-
   sublist(Tail, Sub).

/* sortlists(+List, -Sorted)
      Sort a list of lists ascendingly according to length, deleting the
      empty list if present.
*/
sortlists(List, SortLists) :-
   taglists(List, Tagged),
   sizesort(Tagged, TaggedSorted),
   taglists(TaggedSorted, Sorted),
   delete(Sorted, [], SortLists), !.
```

```
/* sizesort(+TaggedList, -TaggedSorted)
      Sort a list of lists, with the inner lists tagged with their lengths,
      only according to length, retaining the order otherwise.
*/
sizesort(TaggedList, TaggedSorted) :-
   numtags(TaggedList, NumTags),
   sort(NumTags, SortedNums),
   sizesort(SortedNums, TaggedList, NestedTaggedSorted),
   flatten(NestedTaggedSorted, TaggedSorted).
sizesort([], _, []).
sizesort([Num|Tail], TaggedList, [NumTagged|NTail]) :-
   findall(Num:List, member(Num:List,TaggedList), NumTagged),
   sizesort(Tail, TaggedList, NTail).

/* taglists(+List, -TaggedOrUntaggedList)
      Prefixes a list of lists with the lengths of the sublists, or removes
      these prefixes if present.
*/
taglists([], []).

taglists([_:Head|Tail], [Head|TTail]) :-
   taglists(Tail, TTail).
taglists([Head|Tail], [Len:Head|TTail]) :-
   length(Head, Len),
   taglists(Tail, TTail).

/* numtags(+TaggedList, -Numtags)
      Extract numbers from a list tagged with numbers (format is Num:List).
*/
numtags([], []).
numtags([Num:_|Tail], [Num|NTail]) :-
   numtags(Tail, NTail).
```
Listing 1: `causalities.pl`