Dynamic Epistemic Modelling

Jan van Eijck
CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

December 7, 2004

Abstract

This paper introduces *DEMO*, a Dynamic Epistemic Modelling tool. *DEMO* allows modelling epistemic updates, graphical display of update results, graphical display of action models, formula evaluation in epistemic models, translation of dynamic epistemic formulas to PDL formulas, and so on. The paper implements the reduction of dynamic epistemic logic [16, 2, 3, 1] to PDL given in [12]. The reduction of dynamic epistemic logic to automata PDL from [24] is also discussed and implemented. Epistemic models are minimized under bisimulation, and update action models are minimized under action emulation (the appropriate structural notion for having the same update effect, cf. [13]). The paper is an exemplar of tool building for epistemic update logic. It contains the full code of an implementation in Haskell [22], in 'literate programming' style [23], of *DEMO*.

Keywords: Knowledge representation, epistemic updates, dynamic epistemic modelling, action models. information change, logic of communication.

ACM Classification (1998) E 4, F 4.1, H 1.1.

1 A Demo of DEMO

In this introduction we will demonstrate how *DEMO*, which is short for *Dynamic Epistemic MOdelling*,[1] can be used to check semantic intuitions about what goes on in epistemic update situations.[2] For didactic purposes, the initial examples have been kept extremely simple. Although the situation of message passing about just two basic propositions with just three epistemic agents already reveals many subtleties, the reader should bear in mind that *DEMO* is capable of modelling much more complex situations.

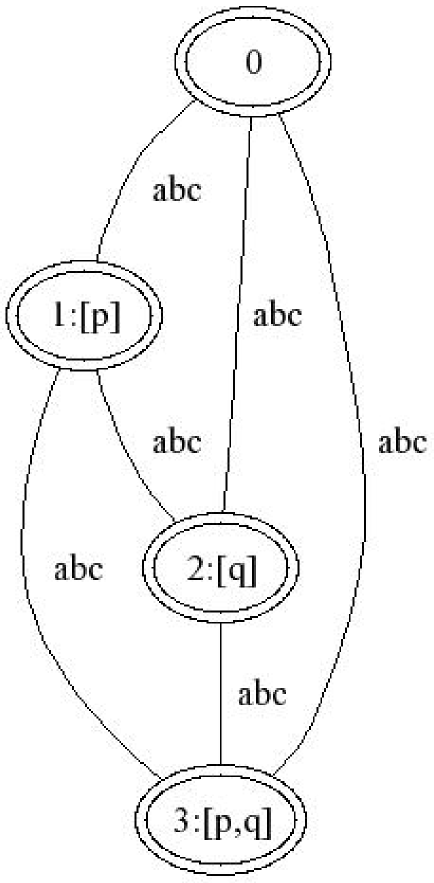
In a situation where you and I know nothing about a particular aspect of the state of the world (about whether p and q hold, say), our state of knowledge is modelled by a Kripke model where the worlds are the four different possibilities for the truth of p and q ($\emptyset$, p, q, pq), your epistemic accessibility relation $\sim_a$ is the total relation on these four possibilities, and mine $\sim_b$ is the total

[1]Or short for *DEMO of Epistemic MOdelling*, for those who prefer co-recursive acronyms.
[2]The program source code is available from `http://www.cwi.nl/~jve/papers/04/demo/`.

relation on these four possibilities as well. There is also c, who like the two of us, is completely ignorant about p and q. This initial model is generated by *DEMO* as follows:

```
DEMO> showM (initE [P 0,Q 0])
==> [0,1,2,3]
[0,1,2,3]
(0,[])(1,[p])(2,[q])(3,[p,q])
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
(c,[[0,1,2,3]])
```

Here is another representation of this same model. This representation can be generated with *dot* [26] from the file produced by the DEMO command `writeP "filename"  (initE [P 0,Q 0])`.



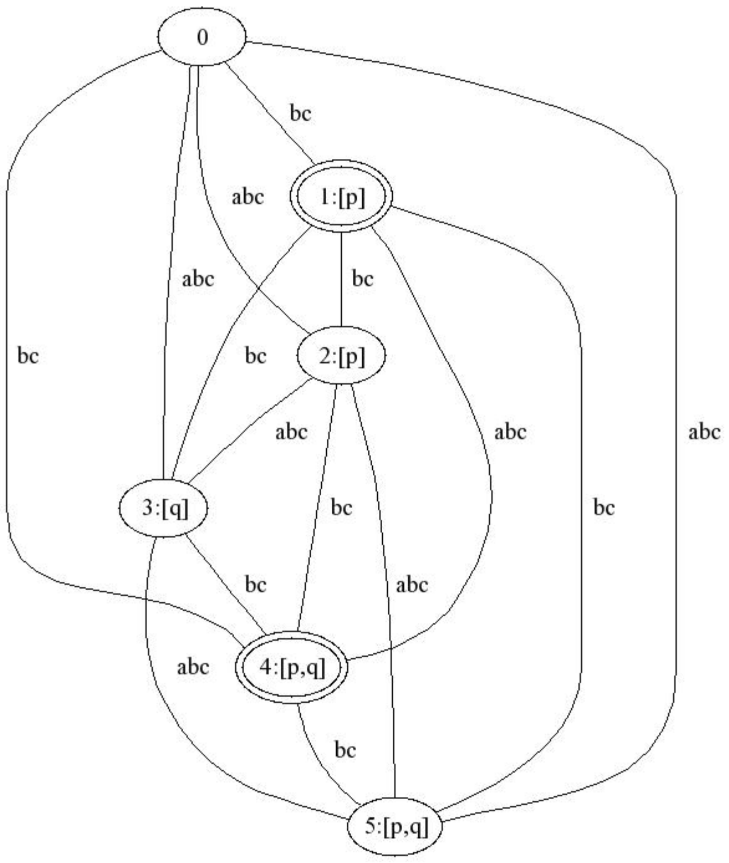
This is a model where none of the three agents a, b or c can distinguish between the four possibilities about p and q. *DEMO* shows the partitions generated by the accessibility relations $\sim_a, \sim_b, \sim_c$. Since these three relations are total, the three partitions each consist of a single block. Call this model `e0`.

Now suppose a wants to know whether p is the case. She asks whether p and receives a truthful answer from somebody who is in a position to know. This answer is conveyed to a in a message. b and c have heard a's question, and so are aware of the fact that an answer may have reached a. b and c have not seen *that* an answer was delivered. This is not a secret communication, for b and c know that a has inquired about p. The situation now changes as follows:

```
DEMO> showM (upd e0 (message a p))
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
(a,[[0,2,3,5],[1,4]])
(b,[[0,1,2,3,4,5]])
(c,[[0,1,2,3,4,5]])
```

This is again a model where the three accessibility relations are equivalences, but one in which a has restricted her range of possibilities to $1, 4$ (these are worlds where p is the case), while for b and c all possibilities are still open.

In graphical display format:



Notice that in this new situation some subtle things have changed for b and c as well. Before the arrival of the message, $\Box_b \neg \Box_a p$ was true, for b knew that a did not know about p. But now b has heard a's question about p, and is aware of the fact that an answer may have reached a. So in the new situation b is not sure anymore about what a knows about p. In other words, $\Box_b \neg \Box_a p$ has become false. On the other hand it is still the case that b knows that a knows nothing about q: $\Box_b \neg \Box_a q$ is still true in the new situation. The situation for c is similar to that for b. These things can be checked in *DEMO* as follows:

```
DEMO> isTrue (upd e0 (message a p)) (K b (Neg (K a q)))
True
DEMO> isTrue (upd e0 (message a p)) (K b (Neg (K a p)))
False
```

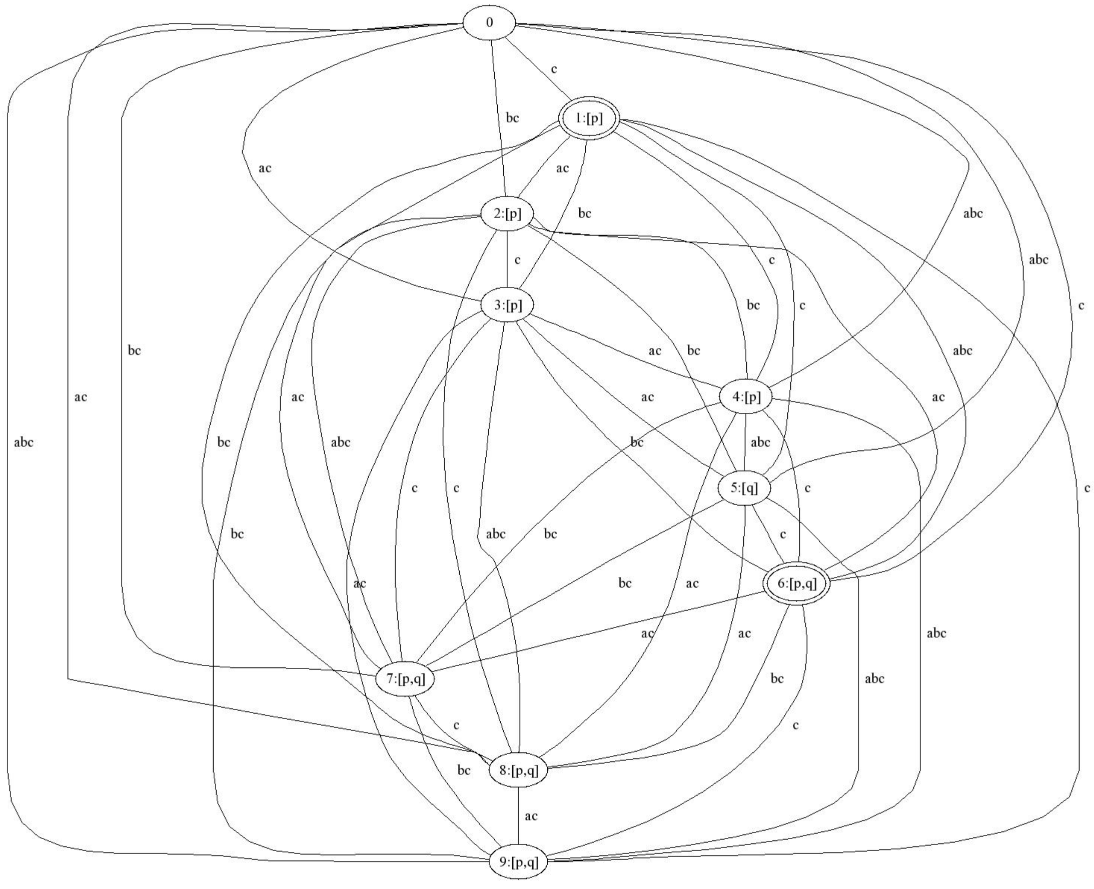
If you receive the same message about p twice, the second time the message gets delivered has no further effect:

```
DEMO> showM (upds e0 [message a p, message a p])
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
(a,[[0,2,3,5],[1,4]])
(b,[[0,1,2,3,4,5]])
(c,[[0,1,2,3,4,5]])
```

Now suppose that the second action is a message informing b about p:

```
DEMO> showM (upds e0 [message a p, message b p])
==> [1,6]
[0,1,2,3,4,5,6,7,8,9]
(0,[])(1,[p])(2,[p])(3,[p])(4,[p])
(5,[q])(6,[p,q])(7,[p,q])(8,[p,q])(9,[p,q])

(a,[[0,3,4,5,8,9],[1,2,6,7]])
(b,[[0,2,4,5,7,9],[1,3,6,8]])
(c,[[0,1,2,3,4,5,6,7,8,9]])
```

The graphical representation of this model is slightly more difficult to fathom at a glance.



In this model a and b both know about p, but they do not know about each other's knowledge about p. c still knows nothing, and both a and b know that c knows nothing. Both $\Box_a\Box_b p$ and $\Box_b\Box_a p$ are false in this model. $\Box_a\neg\Box_b p$ and $\Box_b\neg\Box_a p$ are false as well, but $\Box_a\neg\Box_c p$ and $\Box_b\neg\Box_c p$ are true.

```
EMO> isTrue (upds e0 [message a p, message b p]) (K a (K b p))
False
DEMO> isTrue (upds e0 [message a p, message b p]) (K b (K a p))
False
DEMO> isTrue (upds e0 [message a p, message b p]) (K b (Neg (K b p)))
False
DEMO> isTrue (upds e0 [message a p, message b p]) (K b (Neg (K c p)))
True
```

The order in which a and b are informed does not matter:

```
DEMO> showM (upds e0 [message b p, message a p])
==> [1,6]
[0,1,2,3,4,5,6,7,8,9]
```

```
(0,[])(1,[p])(2,[p])(3,[p])(4,[p])
(5,[q])(6,[p,q])(7,[p,q])(8,[p,q])(9,[p,q])

(a,[[0,2,4,5,7,9],[1,3,6,8]])
(b,[[0,3,4,5,8,9],[1,2,6,7]])
(c,[[0,1,2,3,4,5,6,7,8,9]])
```

Modulo renaming this is the same as the earlier result. The example shows that the epistemic effects of distributed message passing are quite different from those of a public announcement or a group message.

```
DEMO> showM (upd e0 (public p))
==> [0,1]
[0,1]
(0,[p])(1,[p,q])
(a,[[0,1]])
(b,[[0,1]])
(c,[[0,1]])
```

The result of the public announcement that p is that a, b and c are informed that p and about each other's knowledge about p.

DEMO allows to compare the action models for public announcement and individual message passing:

```
DEMO> showM (public p)
==> [0]
[0]
(0,p)
(a,[[0]])
(b,[[0]])
(c,[[0]])

DEMO> showM (cmp [message a p, message b p, message c p])
==> [0]
[0,1,2,3,4,5,6,7]
(0,p)(1,p)(2,p)(3,p)(4,p)
(5,p)(6,p)(7,T)
(a,[[0,1,2,3],[4,5,6,7]])
(b,[[0,1,4,5],[2,3,6,7]])
(c,[[0,2,4,6],[1,3,5,7]])
```

Here `cmp` gives the sequential composition of a list of communicative actions.

More subtly, the situation is also different from a situation where a, b receive the same message that p, with a being aware of the fact that b receives the message and vice versa. Such group messages create common knowledge:

```
DEMO> showM (groupM [a,b] p)
```

```
==> [0]
[0,1]
(0,p)(1,T)
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
```

The difference with the case of the two separate messages is that now a and b are aware of each other's knowledge that p:

```
DEMO> isTrue (upd e0 (groupM [a,b] p)) (K a (K b p))
True
DEMO> isTrue (upd e0 (groupM [a,b] p)) (K b (K a p))
True
```

Next, look at the case where two separate messages reach a and b, one informing a that p and the other informing b that $\neg q$:

```
DEMO> showM (upds e0 [message a p, message b (Neg q)])
==> [2]
[0,1,2,3,4,5,6,7,8]
(0,[])(1,[])(2,[p])(3,[p])(4,[p])
(5,[p])(6,[q])(7,[p,q])(8,[p,q])
(a,[[0,1,4,5,6,8],[2,3,7]])
(b,[[0,2,4],[1,3,5,6,7,8]])
(c,[[0,1,2,3,4,5,6,7,8]])
```

Again the order in which these messages are delivered is immaterial for the end result, as you should expect:

```
DEMO> showM (upds e0 [message b (Neg q), message a p])
==> [2]
[0,1,2,3,4,5,6,7,8]
(0,[])(1,[])(2,[p])(3,[p])(4,[p])
(5,[p])(6,[q])(7,[p,q])(8,[p,q])
(a,[[0,1,3,5,6,8],[2,4,7]])
(b,[[0,2,3],[1,4,5,6,7,8]])
(c,[[0,1,2,3,4,5,6,7,8]])
```

Modulo a renaming of worlds, this is the same as the previous result.

The logic of public announcements and private messages is the so-called logic of knowledge [19]. This logic satisfies the following postulates:

- knowledge distribution $\Box_a(\varphi \Rightarrow \psi) \Rightarrow (\Box_a\varphi \Rightarrow \Box_a\psi)$ (if a knows that φ implies ψ, and she knows φ, then she also knows ψ),
- positive introspection $\Box_a\varphi \Rightarrow \Box_a\Box_a\varphi$ (if a knows φ, then a knows that she knows φ),

- negative introspection $\neg\Box_a \Rightarrow \Box_a\neg\Box_a\varphi$ (if a does not know φ, then she knows that she does not know),
- truthfulness $\Box_a\varphi \Rightarrow \varphi$ (if a knows φ then φ is true).

As is well known, the first of these is valid on all Kripke frames, the second is valid on precisely the transitive Kripke frames, the third is valid on precisely the euclidean Kripke frames (a relation R is euclidean if it satisfies $\forall x\forall y\forall z((xRy \wedge xRz) \Rightarrow yRz))$, and the fourth is valid on precisely the reflexive Kripke frames. A frame satisfies transitivity, euclideanness and reflexivity iff it is an equivalence relation, hence the logic of knowledge is the logic of the so-called S5 Kripke frames: the Kripke frames with an equivalence $\sim_a$ as epistemic accessibility relation. Multi-agent epistemic logic extends this to multi-S5, with an equivalence $\sim_b$ for every $b \in B$, where b is the set of epistemic agents.

Now suppose that instead of open messages, we use *secret* messages. If a secret message is passed to a, b and c are not even aware that any communication is going on. This is the result when a receives a secret message that p in the initial situation:

```
DEMO> showM (upd e0 (secret [a] p))
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
(a,[([],[0,2,3,5]),([],[1,4])])
(b,[([1,4],[0,2,3,5])])
(c,[([1,4],[0,2,3,5])])
```

This is not an S5 model anymore. The accessibility for a is still an equivalence, but the accessibility for b is lacking the property of reflexivity. The worlds $1, 4$ that make up a's conceptual space (for these are the worlds accessible for a from the actual world 1) are precisely the worlds where the b and c arrows are not reflexive. b enters his conceptual space from the vantage point 1, but b does not see the actual world itself. Similarly for c. In the *DEMO* representation, a list `(xs,ys)` gives the entry points `xs` into conceptual space `ys`.

The secret message has no effect on what b and c believe about the facts of the world, but it has effected b's and c's beliefs about the beliefs of a in a disastrous way. These beliefs have become inaccurate. For instance, b now believes that a does *not* know that p, but he is mistaken! The formula $\Box_b\neg\Box_a p$ is true in the actual world, but $\neg\Box_a p$ is false in the actual world, for a *does* know that p, because of the secret message. Here is what *DEMO* says about the situation:

```
DEMO> isTrue (upd e0 (secret [a] p)) (K b (Neg (K a p)))
True
DEMO> isTrue (upd e0 (secret [a] p)) (Neg (K a p))
False
```

This illustrates a regress from the world of knowledge to the world of consistent belief: the result of the update with a secret propositional message does not satisfy the postulate of truthfulness anymore.

The logic of consistent belief satisfies the following postulates:

- knowledge distribution $\Box_a(\varphi \Rightarrow \psi) \Rightarrow (\Box_a\varphi \Rightarrow \Box_a\psi)$,
- positive introspection $\Box_a\varphi \Rightarrow \Box_a\Box_a\varphi$,
- negative introspection $\neg\Box_a \Rightarrow \Box_a\neg\Box_a\varphi$,
- consistency $\Box_a\varphi \Rightarrow \Diamond_a\varphi$ (if a believes that φ then there is a world where φ is true, i.e., φ is consistent).

Consistent belief is like knowledge, except for the fact that it replaces the postulate of truthfulness $\Box_a\varphi \Rightarrow \varphi$ by the weaker postulate of consistency.

Since the postulate of consistency determines the serial Kripke frames (a relation R is serial if $\forall x \exists y\ xRy$), the principles of consistent belief determine the Kripke frames that are transitive, euclidean and serial, the so-called KD45 frames.

In the conceptual world of secrecy, inconsistent beliefs are not far away. Suppose that a, after having received a secret message informing her about p, sends a message to b to the effect that $\Box_a p$. The trouble is that this is *inconsistent* with what b believes.

```
DEMO> showM (upds e0 [secret [a] p, message b (K a p)])
==> [1,5]
[0,1,2,3,4,5,6,7]
(0,[])(1,[p])(2,[p])(3,[p])(4,[q])
(5,[p,q])(6,[p,q])(7,[p,q])
(a,([],[([],[0,3,4,7]),([],[1,2,5,6])]))
(b,([1,5],[([2,6],[0,3,4,7])]))
(c,([],[([1,2,5,6],[0,3,4,7])]))
```

This is not a KD45 model anymore, for it lacks the property of seriality for b's belief relation. b's belief contains two isolated worlds $1, 5$. Since 1 is the actual world, this means that b's belief state has become inconsistent: from now on, b will believe *anything.*

So we have arrived at a still weaker logic. The logic of possibly inconsistent belief satisfies the following postulates:

- knowledge distribution $\Box_a(\varphi \Rightarrow \psi) \Rightarrow (\Box_a\varphi \Rightarrow \Box_a\psi)$,
- positive introspection $\Box_a\varphi \Rightarrow \Box_a\Box_a\varphi$,
- negative introspection $\neg\Box_a \Rightarrow \Box_a\neg\Box_a\varphi$.

This is the logic of K45 frames: frames that are transitive and euclidean.

In [11] some results and a list of questions are given about the possible deterioration of knowledge and belief caused by different kind of message passing. E.g., the result of updating an S5 model with a public announcement or a non-secret message, if defined, is again S5. The result of updating an S5 model with a secret message to some of the agents, if defined, need not even be KD45. One can prove that the result is KD45 iff the model we start out with satisfies certain epistemic conditions. The update result always is K45. Such observations illustrate why S5, KD45 and K45 are ubiquitous in epistemic modelling. See [6, 17] for general background on modal logic, and [7, 14] for specific background on these systems.

If this introduction has convinced the reader that the logic of public announcements, private messages and secret communications is rich and subtle enough to justify the building of the conceptual modelling tools to be presented in the rest of the paper, then it has served is purpose.

In the rest of the paper, we first fix a formal version of epistemic update logic as an implementation goal. After that, we are ready for the implementation.

Some facts about S5, KD45 and K45 relations that are used in the implementation for presenting S5, KD45 and K45 models in a perspicuous way are given in Appendix A.

Further information on various aspects of dynamic epistemic logic is provided in [1, 4, 5, 10, 14, 15, 25].

2 Models and Updates

In this section we formalize the version of dynamic epistemic logic that we are going to implement.

Let p range over a set of basic propositions P and let a range over a set of agents Ag. Then the language of PDL over P, Ag is given by:

$$\begin{array}{rcl} \varphi & ::= & \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid [\pi]\varphi \\ \pi & ::= & a \mid ?\varphi \mid \pi_1;\pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^* \end{array}$$

Employ the usual abbreviations: $\perp$ is shorthand for $\neg\top$, $\varphi_1 \vee \varphi_2$ is shorthand for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ is shorthand for $\neg(\varphi_1 \wedge \varphi_2)$, $\varphi_1 \leftrightarrow \varphi_2$ is shorthand for $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$, and $\langle\pi\rangle\varphi$ is shorthand for $\neg[\pi]\neg\varphi$. Also, if $B \subseteq Ag$ and B is finite, use B as shorthand for $b_1 \cup b_2 \cup \cdots$. Under this convention, the general knowledge operator $E_B\varphi$ takes the shape $[B]\varphi$, while the common knowledge operator $C_B\varphi$ appears as $[B^*]\varphi$, i.e., $[B]\varphi$ expresses that it is general knowledge among agents B that φ, and $[B^*]\varphi$ expresses that it is common knowledge among agents B that φ. In the special case where $B = \emptyset$, B turns out equivalent to $?\perp$, the program that always fails.

The semantics of PDL over P, Ag is given relative to labelled transition systems $\mathbf{M} = (W, V, R)$, where W is a set of worlds (or states), $V : W \rightarrow \mathcal{P}(P)$ is a valuation function, and $R = \{\xrightarrow{a} \subseteq W \times W \mid a \in Ag\}$ is a set of labelled transitions, i.e., binary relations on W, one for each label a. In what follows, we will take the labeled transitions for a to represent the epistemic alternatives of an agent a.

The formulae of PDL are interpreted as subsets of $W_{\mathbf{M}}$ (the state set of $\mathbf{M}$), the actions of PDL as binary relations on $W_{\mathbf{M}}$, as follows:

$$\begin{array}{rcl}
[\![\top]\!]^{\mathbf{M}} & = & W_{\mathbf{M}} \\
[\![p]\!]^{\mathbf{M}} & = & \{w \in W_{\mathbf{M}} \mid p \in V_{\mathbf{M}}(w)\} \\
[\![\neg\varphi]\!]^{\mathbf{M}} & = & W_{\mathbf{M}} - [\![\varphi]\!]^{\mathbf{M}} \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{M}} & = & [\![\varphi_1]\!]^{\mathbf{M}} \cap [\![\varphi_2]\!]^{\mathbf{M}} \\
[\![[\pi]\varphi]\!]^{\mathbf{M}} & = & \{w \in W_{\mathbf{M}} \mid \forall v(\text{ if } (w,v) \in [\![\pi]\!]^{\mathbf{M}} \text{ then } v \in [\![\varphi]\!]^{\mathbf{M}})\} \\
\\
[\![a]\!]^{\mathbf{M}} & = & \xrightarrow{a}_{\mathbf{M}} \\
[\![?\varphi]\!]^{\mathbf{M}} & = & \{(w,w) \in W_{\mathbf{M}} \times W_{\mathbf{M}} \mid w \in [\![\varphi]\!]^{\mathbf{M}}\} \\
[\![\pi_1;\pi_2]\!]^{\mathbf{M}} & = & [\![\pi_1]\!]^{\mathbf{M}} \circ [\![\pi_2]\!]^{\mathbf{M}} \\
[\![\pi_1 \cup \pi_2]\!]^{\mathbf{M}} & = & [\![\pi_1]\!]^{\mathbf{M}} \cup [\![\pi_2]\!]^{\mathbf{M}} \\
[\![\pi^*]\!]^{\mathbf{M}} & = & ([\![\pi]\!]^{\mathbf{M}})^*
\end{array}$$

If $w \in W_{\mathbf{M}}$ then we use $\mathbf{M} \models_w \varphi$ for $w \in [\![\varphi]\!]^{\mathbf{M}}$.

[3] proposes to model epistemic actions as epistemic models, with valuations replaced by preconditions. See also: [4, 5, 10, 12, 14, 15, 25, 29].

Action models for a given language $\mathcal{L}$ Let a set of agents Ag and an epistemic language $\mathcal{L}$ be given. An action model for $\mathcal{L}$ is a triple $A = ([s_0, \ldots, s_{n-1}], \text{pre}, T)$ where $[s_0, \ldots, s_{n-1}]$ is a finite list of action states, $\text{pre} : \{s_0, \ldots, s_{n-1}\} \to \mathcal{L}$ assigns a precondition to each action state, and $T : Ag \to \mathcal{P}(\{s_0, \ldots, s_{n-1}\}^2)$ assigns an accessibility relation $\xrightarrow{a}$ to each agent $a \in Ag$.

A pair $\mathbf{A} = (A, s)$ with $s \in \{s_0, \ldots, s_{n-1}\}$ is a pointed action model, where s is the action that actually takes place.

The list ordering of the action states in an action model will play an important role in the definition of the program transformations associated with the action models.

In the definition of action models, $\mathcal{L}$ can be any language that can be interpreted in PDL models. Actions can be executed in PDL models by means of the following product construction:

Action Update Let a PDL model $\mathbf{M} = (W, V, R)$, a world $w \in W$, and a pointed action model (A, s), with $A = ([s_0, \ldots, s_{n-1}], \text{pre}, T)$, be given. Then the result of executing (A, s) in $(\mathbf{M}, w)$ is the model $(\mathbf{M} \otimes A, (w, s))$, with $\mathbf{M} \otimes A = (W', V', R')$, where

$$\begin{array}{rcl}
W' & = & \{(w,s) \mid s \in \{s_0, \ldots, s_{n-1}\}, w \in [\![\text{pre}(s)]\!]^{\mathbf{M}}\} \\
V'(w,s) & = & V(w) \\
R'(a) & = & \{((w,s),(w',s')) \mid (w,w') \in R(a), (s,s') \in T(a)\}.
\end{array}$$

The language of PDL^{DEL} (update PDL) is given by extending the PDL language with update constructions $[A, s]\varphi$, where (A, s) is a pointed action model. The interpretation of $[A, s]\varphi$ in $\mathbf{M}$ is given by:

$$[\![[A,s]\varphi]\!]^{\mathbf{M}} = \{w \in W_{\mathbf{M}} \mid \text{if } \mathbf{M} \models_w \text{pre}(s) \text{ then } (w,s) \in [\![\varphi]\!]^{\mathbf{M} \otimes A}\}.$$

Using $\langle A, s\rangle\varphi$ as shorthand for $\neg[A, s]\neg\varphi$, we see that the interpretation for $\langle A, s\rangle\varphi$ turns out as:

$$[\![\langle A, s\rangle\varphi]\!]^{\mathbf{M}} = \{w \in W_{\mathbf{M}} \mid \mathbf{M} \models_w \text{pre}(s) \text{ and } (w, s) \in [\![\varphi]\!]^{\mathbf{M}\otimes A}\}.$$

Updating with multiple pointed update actions is also possible. A multiple pointed action is a pair (A, S), with A an action model, and S a subset of the state set of A. Extend the language with updates $[A, S]\varphi$, and interpret this as follows:

$$[\![[A, S]\varphi]\!]^{\mathbf{M}} \quad = \quad \{w \in W_{\mathbf{M}} \mid \forall s \in S(\text{ if } \mathbf{M} \models_w \text{pre}(s) \text{ then } \mathbf{M} \otimes A \models_{(w,s)} \varphi)\}.$$

In [12] it is shown how dynamic epistemic logic can be reduced to PDL by program transformation. Each action model $\mathbf{A}$ has associated program transformers $T_{ij}^{\mathbf{A}}$ for all states s_i, s_j in the action model, such that the following hold:

Lemma 1 (Program Transformation, Van Eijck [12]) *Assume A has n states $s_0, \dots, s_{n-1}$. Then:*

$$\mathbf{M} \models_w [A, s_i][\pi]\varphi \textit{ iff } \mathbf{M} \models_w \bigwedge_{j=0}^{n-1} [T_{ij}^{A}(\pi)][A, s_j]\varphi.$$

This lemma allows a reduction of dynamic epistemic logic to PDL, a reduction that we will implement in the code below.

3 Operations on Action Models

Sequential Composition If $(\mathbf{A}, S)$ and $(\mathbf{B}, T)$ are multiple pointed action models, their sequential composition $(\mathbf{A}, S) \odot (\mathbf{B}, T)$ is given by:

$$(\mathbf{A}, S) \odot (\mathbf{B}, T) := ((W, \text{pre}, R), S \times T),$$

where

- $W = W_{\mathbf{A}} \times W_{\mathbf{B}}$,
- $\text{pre}(s, t) = \text{pre}(s) \wedge \langle \mathbf{A}, S\rangle \text{pre}(t)$,
- R is given by: $(s, t) \xrightarrow{a} (s', t') \in R$ iff $s \xrightarrow{a} s' \in R_{\mathbf{A}}$ and $t \xrightarrow{a} t' \in R_{\mathbf{B}}$.

The unit element for this operation is the action model

$$\mathbf{1} = ((\{0\}, 0 \mapsto \top, \{0 \xrightarrow{a} 0 \mid a \in Ag\}), \{0\}).$$

Updating an arbitary epistemic model $\mathbf{M}$ with $\mathbf{1}$ changes nothing.

Non-deterministic Sum The non-deterministic sum $\oplus$ of multiple-pointed action models $(\mathbf{A}, S)$ and $(\mathbf{B}, T)$ is the action model $(\mathbf{A}, S) \oplus (\mathbf{B}, T)$ is given by:

$$(\mathbf{A}, S) \oplus (\mathbf{B}, T) := ((W, \text{pre}, R), S \uplus T),$$

where $\uplus$ denotes disjoint union, and where

- $W = W_{\mathbf{A}} \uplus W_{\mathbf{B}}$,
- $\text{pre} = \text{pre}_{\mathbf{A}} \uplus \text{pre}_{\mathbf{B}}$,
- $R = R_{\mathbf{A}} \uplus R_{\mathbf{B}}$.

The unit element for this operation is called $\mathbf{0}$: the multiple pointed action model given by $((\emptyset, \emptyset, \emptyset), \emptyset)$.

4 Automata

The reduction of dynamic epistemic logic to PDL from [12] was inspired by a more involved reduction to Automata PDL (PDL, with the atomic programs replaced by nondeterministic finite automata, cf. [18, Chapter 10.3]. That reduction is also implemented below. In this section we fix some terminology about automata.

The general knowledge and common knowledge operators can be encoded as automata. Define a nondeterministic finite automaton or NFA over alphabet Σ as a quadruple consisting of a set of states S, a start state $s \in S$, a set δ of transitions (u, σ, v) with $u \in S, v \in S$, and $\sigma \in \Sigma$, and a final state $f \in S$.

The language accepted by an automaton over Σ is the set of strings from Σ^* that the automaton recognizes (or: accepts), where (S, s, δ, f) recognizes the empty string ϵ iff $s = f$, and (S, s, δ, f) recognizes the string $(\sigma; \overrightarrow{\sigma})$ iff there is a $u \in S$ such that $(s, \sigma, u) \in \delta$ and (S, u, δ, f) recognizes $\overrightarrow{\sigma}$. If $\overrightarrow{\sigma}$ is accepted by Aut, we say that $\overrightarrow{\sigma} \in \text{Aut}$.

A NFA N with set of accept states F can be modelled as the set of NFAs $\{N_f \mid f \in F\}$, where N_f is like N except for the fact that it has a single final state f.

Some example automata that are relevant in the present context:

- The automaton for general knowledge among agents B is the automaton with start state 0, final state 1, and transitions $\{(0, b, 1) \mid b \in B\}$.
- The automaton for common knowledge among agents B is the automaton with start state 0, final state 0 and transitions $\{(0, b, 0) \mid b \in B\}$.
- The automaton for common knowledge relativised to φ among agents B (see [24]) is the automaton with start state 0, final state 1, and transitions $\{(0, ?\varphi, 1)\} \cup \{(1, b, 0) \mid b \in B\}$.

Following [24], we extend the language $\mathcal{LANG}$ with formulas $[\text{Aut}]\varphi$, where Aut is an automaton.

Let $\Sigma = A \cup \{?\varphi \mid \varphi \in \mathcal{LANG}\}$. Define the interpretation of a string $\overrightarrow{\sigma}$ from Σ^* in a model M as follows:

$$\begin{array}{rcl} [\![\epsilon]\!]^M & = & \{(w,w) \mid w \in W_M\} \\ [\![a;\overrightarrow{\sigma}]\!]^M & = & \xrightarrow{a}_M \circ [\![\overrightarrow{\sigma}]\!]^M \\ [\![?\varphi;\overrightarrow{\sigma}]\!]^M & = & \{(w,w) \mid M \models_w \varphi\} \circ [\![\overrightarrow{\sigma}]\!]^M \end{array}$$

The truth definition for $[\text{Aut}]\varphi$ is now given by:

$$\begin{array}{rcl} M \models_w [\text{Aut}]\varphi & :\equiv & \text{for all } v \in W_M \text{ and all } \overrightarrow{\sigma} \in \text{Aut} : \\ & & \quad \text{if } (w,v) \in [\![\overrightarrow{\sigma}]\!]^M \text{ then } M \models_v \varphi. \end{array}$$

As it stands, it is not immediately clear how to implement this as a finite check, as both the list of strings accepted by an automaton and the list of paths between pairs of points in a model are generally infinite. In Section 18 we propose a modification of a graph reachability algorithm that computes the sets of worlds reachable from a given world in a model through a path accepted by a given automaton.

In the spirit of [24] (but with a slight modification), define a function AUT from quadruples consisting of an action model $\mathbf{A}$, a first state in $\mathbf{A}$, a second state in $\mathbf{A}$, and an automaton $\text{Aut} = (S, s, \delta, f)$, to automata, as follows:

$$\text{AUT}(\mathbf{A}, \mathbf{s}, \mathbf{t}, (S, s, \delta, f)) = (S', s', \delta', f')$$

where

$$S' = S_{\mathbf{A}} \times \{0,1\} \times S,$$

$$s' = (\mathbf{s}, 0, s),$$

$$f' = (\mathbf{t}, 1, f),$$

$$\begin{array}{rcl} \delta' & = & \{((\mathbf{u},1,t), a, (\mathbf{u}',0,t')) \mid \mathbf{u} \xrightarrow{a} \mathbf{u}', (t,a,t') \in \delta\} \\ & \cup & \{((\mathbf{u},0,t), ?\varphi, (\mathbf{u},1,t)) \mid \mathbf{u} \in S_{\mathbf{A}}, p_{\mathbf{u}} = \varphi, t \in S\} \\ & \cup & \{((\mathbf{u},1,t), ?\langle \mathbf{A}, u\rangle\varphi, (\mathbf{u},1,t')) \mid \mathbf{u} \in S_{\mathbf{A}}, (t, ?\varphi, t') \in \delta\}. \end{array}$$

Then one can prove the following Lemma:

Lemma 2 (Reduction; Kooi and Van Benthem)

$$[\mathbf{A}, \mathbf{s}][(S, s, \delta, f)]\varphi \leftrightarrow \bigwedge_{\mathbf{t} \in S_{\mathbf{A}}} [AUT(\mathbf{A}, \mathbf{s}, \mathbf{t}, (S, s, \delta, f))][\mathbf{A}, \mathbf{t}]\varphi.$$

This lemma allows one to reduce epistemic action logic to APDL, the logic of PDL over automata [18, Chapter 10.3].

5 Logics for Communication

Here are some specific action models that can be used to define various languages of communication.

Public announcement of φ: action model $(\mathbf{S}, \{0\})$, with

$$S_{\mathbf{S}} = \{0\}, p_{\mathbf{S}} = 0 \mapsto \varphi, R_{\mathbf{S}} = \{0 \xrightarrow{a} 0 \mid a \in A\}.$$

Individual message to b that φ: action model $(\mathbf{S}, \{0\})$, with

$$S_{\mathbf{S}} = \{0,1\}, p_{\mathbf{S}} = 0 \mapsto \varphi, 1 \mapsto \top, R_{\mathbf{S}} = \{0 \sim_a 1 \mid a \in A - \{b\}\}.$$

Group message to B that φ: action model $(\mathbf{S}, \{0\})$, with

$$S_{\mathbf{S}} = \{0,1\}, p_{\mathbf{S}} = 0 \mapsto \varphi, 1 \mapsto \top, R_{\mathbf{S}} = \{0 \sim_a 1 \mid a \in A - B\}.$$

Secret individual communication to b that φ: action model $(\mathbf{S}, \{0\})$, with

$$\begin{aligned} S_{\mathbf{S}} &= \{0,1\}, \\ p_{\mathbf{S}} &= 0 \mapsto \varphi, 1 \mapsto \top, \\ R_{\mathbf{S}} &= \{0 \xrightarrow{b} 0\} \cup \{0 \xrightarrow{a} 1 \mid a \in A - \{b\}\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\}. \end{aligned}$$

Secret group communication to B that φ: action model $(\mathbf{S}, \{0\})$, with

$$\begin{aligned} S_{\mathbf{S}} &= \{0,1\}, \\ p_{\mathbf{S}} &= 0 \mapsto \varphi, 1 \mapsto \top, \\ R_{\mathbf{S}} &= \{0 \xrightarrow{b} 0 \mid b \in B\} \cup \{0 \xrightarrow{a} 1 \mid a \in A - B\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\}. \end{aligned}$$

Test of φ: action model $(\mathbf{S}, \{0\})$, with

$$S_{\mathbf{S}} = \{0,1\}, p_{\mathbf{S}} = 0 \mapsto \varphi, 1 \mapsto \top, R_{\mathbf{S}} = \{0 \xrightarrow{a} 1 \mid a \in A\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\}.$$

Individual revelation to b of a choice from $\{\varphi_1, \dots, \varphi_n\}$: action model $(\mathbf{S}, \{1, \dots, n\})$, with

$$\begin{aligned} S_{\mathbf{S}} &= \{1, \dots, n\}, \\ p_{\mathbf{S}} &= 1 \mapsto \varphi_1, \dots, n \mapsto \varphi_n, \\ R_{\mathbf{S}} &= \{s \xrightarrow{b} s \mid s \in S_{\mathbf{S}}\} \cup \{s \xrightarrow{a} s' \mid s, s' \in S_{\mathbf{S}}, a \in A - \{b\}\}. \end{aligned}$$

Group revelation to B of a choice from $\{\varphi_1, \dots, \varphi_n\}$: action model $(\mathbf{S}, \{1, \dots, n\})$, with

$$\begin{aligned} S_{\mathbf{S}} &= \{1, \dots, n\}, \\ p_{\mathbf{S}} &= 1 \mapsto \varphi_1, \dots, n \mapsto \varphi_n, \\ R_{\mathbf{S}} &= \{s \xrightarrow{b} s \mid s \in S_{\mathbf{S}}, b \in B\} \cup \{s \xrightarrow{a} s' \mid s, s' \in S_{\mathbf{S}}, a \in A - B\}. \end{aligned}$$

Transparant informedness of B about φ: action model $(\mathbf{S}, \{0,1\})$, with

$$\begin{aligned} S_{\mathbf{S}} &= \{0,1\}, \\ p_{\mathbf{S}} &= 0 \mapsto \varphi, 1 \mapsto \neg\varphi, \\ R_{\mathbf{S}} &= \{0 \xrightarrow{a} 0 \mid a \in A\} \cup \{0 \xrightarrow{a} 1 \mid a \in A - B\} \cup \{1 \xrightarrow{a} 0 \mid a \in A - B\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\}. \end{aligned}$$

Transparant informedness of B about φ is the special case of a group revelation ot B of a choice from $\{\varphi, \neg\varphi\}$. Note that all but the revelation action models and the transparant informedness action models are single pointed (their sets of actual states are singletons).

The language for the logic of public announcements:

$$\begin{array}{rcl}\varphi & ::= & \top \mid p \mid \neg\varphi \mid \bigwedge[\varphi_1,\ldots,\varphi_n] \mid \bigvee[\varphi_1,\ldots,\varphi_n] \mid \Box_a\varphi \mid E_B\varphi \mid C_B\varphi \mid \ [\pi]\varphi \\ \pi & ::= & \mathbf{1} \mid \mathbf{0} \mid \text{public } B\ \varphi \mid \bigodot[\pi_1,\ldots,\pi_n] \mid \bigoplus[\pi_1,\ldots,\pi_n]\end{array}$$

Semantics for this: use the semantics of **1**, **0**, **public** B φ, and the operations on multiple pointed action models from Section 3.

The logic of tests and public announcements: as above, but now also allowing tests $?\varphi$ as basic programs. Semantics: add the semantics of $?\varphi$ to the above repertoire.

The logic of individual messages: as above, but now the basic actions are messages to individual agents. Semantics: start out from the semantics of **message** a φ.

The logic of tests, public announcements, and group revelations as above, but now also allowing revelations from alternatives. Semantics: use the semantics of **reveal** B $\{\varphi_1,\ldots,\varphi_n\}$.

And so on.

6 Module Declaration

We now turn to the implementation.

```
module DEMO
where

import List
import Char
import DPLL
```

Here `List` is a standard Haskell module. `DPLL` is a module for propositional reasoning with the Davis, Putnam, Logemann, Loveland procedure [8, 9] listed in Appendix B.

7 Version

The first version of *DEMO* was written in March 2004. This first version was extended in May 2004 with an implementation of automata and a translation function from epistemic update logic to Automata PDL. In September 2004, I discovered a direct reduction of epistemic update logic to PDL [12]. This motivated a switch to a PDL-like language, with extra modalities for action update and automata update. I decided to leave in the automata for the time being, for nostalgic reasons.

Also in September 2004, Ji Ruan and I found a simple definition for action emulation and an algorithm for computing emulation-minimal action models [13, 29], so reduction under action emulation is also implemented in this October version of DEMO.

```
version :: String
version = "DEMO 1.02, October 2004"
```

8 Agents and Formulas

Agents:

```
data Agent = A | B | C | D | E deriving (Eq,Ord,Enum)
```

Give the agents appropriate names:

```
a, alice, b, bob, c, carol, d, dave, e, ernie  :: Agent
a = A; alice = A
b = B; bob   = B
c = C; carol = C
d = D; dave  = D
e = E; ernie = E
```

Make agents showable in an appropriate way:

```
instance Show Agent where
  show A = "a"; show B = "b"; show C = "c"; show D = "d" ; show E = "e"
```

A function for listing all agents (uncomment the appropriate definition of `last_agent`, depending on the number of agents you need):

```
all_agents :: [Agent]
all_agents = [a .. last_agent]

last_agent :: Agent
--last_agent = a
--last_agent = b
last_agent = c
--last_agent = d
--last_agent = e
```

Basic propositions:

```
data Prop = P Int | Q Int | R Int deriving (Eq,Ord)
```

Show these in the standard way, in lower case, with index 0 omitted.

```
instance Show Prop where
  show (P 0) = "p"; show (P i) = "p" ++ show i
  show (Q 0) = "q"; show (Q i) = "q" ++ show i
  show (R 0) = "r"; show (R i) = "r" ++ show i
```

Formulas, according to the definition:

$$
\begin{array}{lcl}
\varphi & ::= & \top \mid p \mid \neg\varphi \mid \bigwedge[\varphi_1, \ldots, \varphi_n] \mid \bigvee[\varphi_1, \ldots, \varphi_n] \mid [\pi]\varphi \mid \ [\mathbf{A}]\varphi \mid \ [\mathrm{Aut}]\varphi \\
\pi & ::= & a \mid B \mid ?\varphi \mid \bigcirc[\pi_1, \ldots, \pi_n] \mid \bigcup[\pi_1, \ldots, \pi_n] \mid \pi^*
\end{array}
$$

Here, p ranges over basic propositions, a ranges over agents, B ranges over non-empty sets of agents, $\mathbf{A}$ is a multiple pointed action model (see below), and Aut is an automaton. $\bigcirc$ denotes sequential composition of a list of programs. We will often write $\bigcirc[\pi_1, \pi_2]$ as $\pi_1; \pi_2$, and $\bigcup[\pi_1, \pi_2]$ as $\pi_1 \cup \pi_2$.

Note that general knowledge among agents B that φ is expressed in this language as $[B]\varphi$, and common knowledge among agents B that φ as $[B^*]\varphi$. Thus, $[B]\varphi$ can be viewed as shorthand for $[\bigcup_{b\in B} b]\varphi$. In case $B = \emptyset$, $[B]\varphi$ turns out to be equivalent to $[?\bot]\varphi$.

For convenience, we have also left in the more traditional way of expressing individual knowledge $\Box_a\varphi$, general knowledge $E_B\varphi$ and common knowledge $C_B\varphi$.

```
data Form = Top
          | Prop Prop
          | Neg  Form
          | Conj [Form]
          | Disj [Form]
          | Pr Program Form
          | K Agent Form
          | EK [Agent] Form
          | CK [Agent] Form
          | Up PoAM Form
          | Aut (NFA State) Form
          deriving (Eq,Ord)
```

```
data Program = Ag Agent
             | Ags [Agent]
             | Test Form
             | Conc [Program]
             | Sum  [Program]
             | Star Program
             deriving (Eq,Ord)
```

A useful abbreviation:

```
impl :: Form -> Form -> Form
impl form1 form2 = Disj [Neg form1, form2]
```

Show formulas in the standard way:

```
instance Show Form where
  show Top = "T" ; show (Prop p) = show p; show (Neg f) = '-':(show f);
  show (Conj fs)     = '&': show fs
  show (Disj fs)     = 'v': show fs
  show (Pr p f)      = '[': show p ++ "]" ++ show f
  show (K agent f)   = '[': show agent ++ "]" ++ show f
  show (EK agents f) = 'E': show agents ++ show f
  show (CK agents f) = 'C': show agents ++ show f
  show (Up pam f)    = 'A': show (points pam) ++ show f
  show (Aut aut f)   = '[': show aut ++ "]" ++ show f
```

Show programs in a standard way:

```
instance Show Program where
  show (Ag a)       = show a
  show (Ags as)     = show as
  show (Test f)     = '?': show f
  show (Conc ps)    = 'C': show ps
  show (Sum ps)     = 'U': show ps
  show (Star p)     = '(': show p ++ ")*"
```

Programs can get very unwieldy very quickly. As is well known, there is no normalisation procedure for regular expressions. Still, here are some rewriting steps for simplification of programs:

$$
\begin{array}{rcl}
\emptyset & \rightarrow & ?\bot \\
?\varphi_1 \cup ?\varphi_2 & \rightarrow & ?(\varphi_1 \vee \varphi_2) \\
?\bot \cup \pi & \rightarrow & \pi \\
\pi \cup ?\bot & \rightarrow & \pi \\
\bigcup[\pi_1, \ldots, \pi_k, \bigcup[\pi_{k+1}, \ldots, \pi_{k+m}], \pi_{k+m+1}, \ldots, \pi_{k+m+n}] & \rightarrow & \bigcup[\pi_1, \ldots, \pi_{k+m+n}] \\
\bigcup[] & \rightarrow & ?\bot \\
\bigcup[\pi] & \rightarrow & \pi \\
?\varphi_1; ?\varphi_2 & \rightarrow & ?(\varphi_1 \wedge \varphi_2) \\
?\top; \pi & \rightarrow & \pi \\
\pi; ?\top & \rightarrow & \pi \\
?\bot; \pi & \rightarrow & ?\bot \\
\pi; ?\bot & \rightarrow & ?\bot \\
\bigcirc[\pi_1, \ldots, \pi_k, \bigcirc[\pi_{k+1}, \ldots, \pi_{k+m}], \pi_{k+m+1}, \ldots, \pi_{k+m+n}] & \rightarrow & \bigcirc[\pi_1, \ldots, \pi_{k+m+n}] \\
\bigcirc[] & \rightarrow & ?\top \\
\bigcirc[\pi] & \rightarrow & \pi \\
(?\varphi)^* & \rightarrow & ?\top \\
(?\varphi \cup \pi)^* & \rightarrow & \pi^* \\
(\pi \cup ?\varphi)^* & \rightarrow & \pi^* \\
\pi^{**} & \rightarrow & \pi^*
\end{array}
$$

Simplifying unions by splitting up in test part, accessibility part and rest:

```
splitU :: [Program] -> ([Form],[Agent],[Program])
splitU [] = ([],[],[])
splitU (Test f: ps) = (f:fs,ags,prs)
  where (fs,ags,prs) = splitU ps
splitU (Ag x: ps) = (fs,union [x] ags,prs)
  where (fs,ags,prs) = splitU ps
splitU (Ags xs: ps) = (fs,union xs ags,prs)
  where (fs,ags,prs) = splitU ps
splitU (Sum ps: ps') = splitU (union ps ps')
splitU (p:ps)        = (fs,ags,p:prs)
  where (fs,ags,prs) = splitU ps
```

Simplifying compositions:

```
comprC :: [Program] -> [Program]
comprC [] = []
comprC (Test Top: ps)          = comprC ps
comprC (Test (Neg Top): ps)    = [Test (Neg Top)]
comprC (Test f: Test f': rest) = comprC (Test (canonF (Conj [f,f'])): rest)
comprC (Conc ps : ps')         = comprC (ps ++ ps')
comprC (p:ps)                  = let ps' = comprC ps
                                  in
                                    if ps' == [Test (Neg Top)]
                                      then [Test (Neg Top)]
                                      else p: ps'
```

Use this in the code for program simplification:

```
simpl :: Program -> Program
simpl (Ag x)                     = Ag x
simpl (Ags [])                   = Test (Neg Top)
simpl (Ags [x])                  = Ag x
simpl (Ags xs)                   = Ags xs
simpl (Test f)                   = Test (canonF f)
```

Simplifying unions:

```
simpl (Sum prs) =
  let (fs,xs,rest) = splitU (map simpl prs)
      f            = canonF (Disj fs)
  in
    if xs == [] && rest == []
      then Test f
    else if xs == [] && f == Neg Top && length rest == 1
      then (head rest)
    else if xs == [] && f == Neg Top
      then Sum rest
    else if xs == []
      then Sum (Test f: rest)
    else if length xs == 1  && f == Neg Top
      then Sum (Ag (head xs): rest)
    else if length xs == 1
      then Sum (Test f: Ag (head xs): rest)
    else if f == Neg Top
      then Sum (Ags xs: rest)
    else Sum (Test f: Ags xs: rest)
```

Simplifying sequential compositions:

```
simpl (Conc prs) =
    let prs' = comprC (map simpl prs)
    in
      if prs'== []                  then Test Top
      else if length prs' == 1      then head prs'
      else if head prs' == Test Top then Conc (tail prs')
      else                               Conc prs'
```

Simplifying stars:

```
simpl (Star pr) = case simpl pr of
    Test f             -> Test Top
    Sum [Test f, pr']  -> Star pr'
    Sum (Test f: prs') -> Star (Sum prs')
    Star pr'           -> Star pr'
    pr'                -> Star pr'
```

Property of being a purely propositional formula:

```
pureProp ::  Form -> Bool
pureProp Top       = True
pureProp (Prop _)  = True
pureProp (Neg f)   = pureProp f
pureProp (Conj fs) = and (map pureProp fs)
pureProp (Disj fs) = and (map pureProp fs)
pureProp  _        = False
```

Some example formulas and formula-forming operators:

```
bot, p, q, r, p1, p2, p3, q1, q2, q3, r1, r2, r3 :: Form
bot = Neg Top
p   = Prop (P 0); q  = Prop (Q 0); r  = Prop (R 0)
p1  = Prop (P 1); p2 = Prop (P 2); p3 = Prop (P 3)
q1  = Prop (Q 1); q2 = Prop (Q 2); q3 = Prop (Q 3)
r1  = Prop (R 1); r2 = Prop (R 2); r3 = Prop (R 3)
u   = Up ::  PoAM -> Form -> Form

nkap = Neg (K a p)
nkanp = Neg (K a (Neg p))
nka_p = Conj [nkap,nkanp]
```

9 Models and Generated Submodels

Action models are built from states. We assume states are represented by integers:

```
type State = Integer
```

Action models consist of accessibility relations for all agents, a precondition function assigning a precondition in some language to every action, and a designated state *s*. The only difference between an action model and a static model is in the fact that action models have a precondition function that may assign something different than sets of basic propositions.

For the model update construction, it will prove useful to generalize over states. We first define general models, and then specialize to action models and static models. In the following definition, `state` and `formula` are variables over types (whereas in the declaration above, `State` is a concrete type).

```
data Model state formula = Mo
                            [state]
                            [(state,formula)]
                            [(Agent,state,state)]
                            deriving (Eq,Ord,Show)
```

Model with a number of singled-out points:

```
data Pmod state formula = Pmod
                            [state]
                            [(state,formula)]
                            [(Agent,state,state)]
                            [state]
                            deriving (Eq,Ord,Show)
```

Creating a pointed model from a model and a list of points:

```
mod2pmod :: Model state formula -> [state] -> Pmod state formula
mod2pmod (Mo states prec accs) points = Pmod states prec accs points
```

Separating a pointed model into model and points:

```
pmod2mp :: Pmod state formula -> (Model state formula, [state])
pmod2mp (Pmod states prec accs points)  = (Mo states prec accs, points)
```

Decomposing a pointed model into a list of pairs (m,w):

```
decompose ::  Pmod state formula -> [(Model state formula, state)]
decompose (Pmod states prec accs points) =
   [(Mo states prec accs, point) | point <- points ]
```

It is useful to be able to map the precondition table to a function. Here is general tool for that. Note that the resulting function is partial; if the function argument does not occur in the table, the value is undefined.

```
table2fct :: Eq a => [(a,b)] -> a -> b
table2fct t = \ x -> maybe undefined id (lookup x t)
```

The *domain* of a model is its list of states:

```
domain :: Model state formula -> [state]
domain (Mo states _ _) = states
```

The *eval* of a model is its list of state/formula pairs:

```
eval :: Model state formula -> [(state,formula)]
eval (Mo _ pre _) = pre
```

The *access* of a model is its labelled transition component:

```
access :: Model state formula -> [(Agent,state,state)]
access (Mo _ _ rel) = rel
```

The points of a Pmod:

```
points :: Pmod state formula -> [state]
points (Pmod _ _ _ pnts) = pnts
```

When we are looking at pointed models, we are only interested in generated submodels, with as their domain the designated state(s) plus everything that is reachable by an accessibility path.

```
gsm :: Ord state => Pmod state formula ->  Pmod state formula
gsm (Pmod states pre rel points) = (Pmod states' pre' rel' points)
   where
   states' = closure rel all_agents points
   pre'    = [(s,f)     | (s,f)     <- pre,
                          elem s states'                  ]
   rel'    = [(ag,s,s') | (ag,s,s') <- rel,
                          elem s states',
                          elem s' states'                 ]
```

The closure of a state list, given a relation and a list of agents:

```
closure ::  Ord state =>
             [(Agent,state,state)] -> [Agent] -> [state] -> [state]
closure rel agents xs
  | xs' == xs = xs
  | otherwise = closure rel agents xs'
    where
    xs' = (nub . sort) (xs ++ (expand rel agents xs))
```

The expansion of a relation R given a state set S and a set of agents B is given by $\{t \mid s \xrightarrow{b} t \in R, s \in S, b \in B\}$. Implementation:

```
expand :: Ord state =>
           [(Agent,state,state)] -> [Agent] -> [state] -> [state]
expand rel agnts ys =
      (nub . sort . concat)
         [ alternatives rel ag state | ag    <- agnts,
                                       state <- ys       ]
```

The epistemic alternatives for agent a in state s are the states in sR_a (the states reachable through R_a from s):

```
alternatives :: Eq state =>
                 [(Agent,state,state)] -> Agent -> state -> [state]
alternatives rel ag current =
  [ s' | (a,s,s') <- rel, a == ag, s == current ]
```

Static models are models where the states are of type `State`, and the precondition function assigns lists of basic propositions (this specializes the precondition function to a valuation).

```
type SM = Model State [Prop]
```

Epistemic models are static models with a set of distinguished points:

```
type EpistM = Pmod State [Prop]
```

Action models are models where the states are of type `State`, and the precondition function assigns objects of type `Form`.

```
type AM = Model State Form
```

Pointed action models:

```
type PoAM = Pmod State Form
```

The preconditions of a pointed action model:

```
preconditions :: PoAM -> [Form]
preconditions (Pmod states pre acc points) =
   map (table2fct pre) points
```

Sometimes we need a single precondition:

```
precondition :: PoAM -> Form
precondition am = canonF (Conj (preconditions am))
```

The purpose of action models is to define relations on the class of all static models. States with precondition $\perp$ can be pruned from an action model. For this we define a specialized version of the `gsm` function:

```
gsmPoAM :: PoAM -> PoAM
gsmPoAM (Pmod states pre acc points) =
  let
    points' = [ p | p <- points, consistent (table2fct pre p) ]
    states' = [ s | s <- states, consistent (table2fct pre s) ]
    pre'    = filter (\ (x,_) -> elem x states') pre
    f       = \ (_,s,t) -> elem s states' && elem t states'
    acc'    = filter f acc
  in
  if points' == []
     then (Pmod [] [] [] [])
     else gsm (Pmod states' pre' acc' points')
```

10 Representing Accessibility Relations

Formal background for this section is in Appendix A.

Filter out the accessibility relation for a particular agent label:

```
accFor :: Eq a => a -> [(a,b,b)] -> [(b,b)]
accFor label triples = [ (x,y) | (label',x,y) <- triples, label == label' ]
```

An implementation of $\subseteq$ for lists:

```
containedIn :: Eq a => [a] -> [a] -> Bool
containedIn [] ys     = True
containedIn (x:xs) ys = elem x ys && containedIn xs ys
```

The smallest reflexive relation on a list:

```
idR :: Eq a => [a] -> [(a,a)]
idR = map (\x -> (x,x))
```

Test for reflexivity of a relation.

```
reflR :: Eq a => [a] -> [(a,a)] -> Bool
reflR xs r = containedIn (idR xs) r
```

Test for symmetry of a relation:

```
symR :: Eq a => [(a,a)] -> Bool
symR [] = True
symR ((x,y):pairs) | x == y    = symR (pairs)
                   | otherwise = elem (y,x) pairs
                                 && symR (pairs \\ [(y,x)])
```

A check for transitivity of R tests for each couple of pairs $(x, y) \in R, (u, v) \in R$ whether $(x, v) \in R$ if $y = u$:

```
transR :: Eq a => [(a,a)] -> Bool
transR [] = True
transR s = and [ trans pair s | pair <- s ]
   where
   trans (x,y) r = and [ elem (x,v) r | (u,v) <- r, u == y ]
```

Put these together in a test for equivalence:

```
equivalenceR :: Eq a => [a] -> [(a,a)] -> Bool
equivalenceR xs r = reflR xs r && symR r && transR r
```

Checking whether a model is S5:

```
isS5 :: (Eq a) => [a] -> [(Agent,a,a)] -> Bool
isS5 xs triples =
  all (equivalenceR xs) rels
    where rels = [ accFor i triples | i <- all_agents ]
```

From a relation as a list of pairs to a characteristic function:

```
pairs2rel :: (Eq a, Eq b) => [(a,b)] -> a -> b -> Bool
pairs2rel pairs = \ x y -> elem (x,y) pairs
```

Using an equivalence relation to create a partition:

```
rel2part :: (Eq a) => [a] -> (a -> a -> Bool) -> [[a]]
rel2part [] r = []
rel2part (x:xs) r = xblock : rel2part rest r
  where
  (xblock,rest) = partition (\ y -> r x y) (x:xs)
```

From an equivalence relation (represented as a list of pairs) to the corresponding partition:

```
equiv2part :: Eq a => [a] -> [(a,a)] -> [[a]]
equiv2part xs r = rel2part xs (pairs2rel r)
```

One way to test whether a model is KD45 is by means of tests for euclideanness, transitivity and seriality. The test for euclideanness is a variation on the test for transitivity.

```
euclideanR :: Eq a => [(a,a)] -> Bool
euclideanR s = and [ eucl pair s | pair <- s ]
  where
  eucl (x,y) r = and [ elem (y,v) r | (u,v) <- r, u == x ]
```

The test for seriality:

```
serialR :: Eq a => [a] -> [(a,a)] -> Bool
serialR [] s = True
serialR (x:xs) s = any (\ p -> (fst p) == x) s && serialR xs s
```

The test for KD45:

```
kd45R :: Eq a => [a] -> [(a,a)] -> Bool
kd45R xs r = transR r && serialR xs r && euclideanR r
```

A test for K45:

```
k45R :: Eq a => [(a,a)] -> Bool
k45R r = transR r && euclideanR r
```

Test for being an isolated point:

```
isolated  :: Eq a =>  [(a,a)] -> a -> Bool
isolated r x = notElem x (map fst r ++ map snd r)
```

Checking for K45; if successful, return a list of isolated points and a list of balloons (see below).

```
k45PointsBalloons :: Eq a => [a] -> [(a,a)] -> Maybe ([a],[([a],[a])])
k45PointsBalloons xs r =
  let
     orphans = filter (isolated r) xs
     ys = xs \\ orphans
  in
    case kd45Balloons ys r of
      Just balloons -> Just (orphans,balloons)
      Nothing       -> Nothing
```

The entry pairs of a relation:

```
entryPair :: Eq a => [(a,a)] -> (a,a) -> Bool
entryPair r = \ (x,y) -> notElem (y,x) r
```

Checking for KD45 by testing the non-entry pairs for equivalence. If successful, the function returns just a list of balloons, where a balloon is a list pair, with the first element the entry points into the states in the second element.

```
kd45Balloons :: Eq a => [a] -> [(a,a)] -> Maybe [([a],[a])]
kd45Balloons xs r =
  let
     (s,t)          = partition (entryPair r) r
     entryPoints    = map fst s
     nonentryPoints = xs \\ entryPoints
     s5part xs r = if equivalenceR xs r
                   then Just (equiv2part xs t)
                   else Nothing
  in
    case s5part nonentryPoints t of
      Just part ->
        Just [ (nub (map fst (filter (\ (x,y) -> elem y block) s)),
                block) |     block <- part                            ]
      Nothing   ->
        Nothing
```

This gives, e.g.:

```
DEMO>  kd45Balloons [0,1,2] [(1,1),(1,2),(2,1),(2,2),(0,1)]
Just [([0],[1,2])]
DEMO> kd45Balloons [0,1,2,3] [(1,1),(1,2),(2,1),(2,2),(0,1),(3,1)]
Just [([0,3],[1,2])]
DEMO> kd45Balloons [0,1,2,3] [(1,1),(1,2),(2,1),(2,2),(0,1),(3,1),(1,3)]
Nothing
```

If the accessibility relations in a list of triples are all K45, just return the corresponding isolated points plus balloons for each agent. Otherwise return nothing.

```
k45 :: (Eq a, Ord a) => [a] ->
    [(Agent,a,a)] -> Maybe [(Agent,([a],[([a],[a])]))]
k45 xs triples =
  if and [ maybe False (\ x -> True) b | (a,b) <- results  ]
     then Just [ (a, maybe undefined id b) | (a,b) <- results  ]
     else Nothing
       where rels     = [ (a, accFor a triples)  | a     <- all_agents ]
             results  = [ (a, k45PointsBalloons xs r) | (a,r) <- rels   ]
```

If the accessibility relations in a list of triples are all KD45, just return the corresponding balloons for each agent. Otherwise return nothing.

```
kd45 :: (Eq a, Ord a) => [a] -> [(Agent,a,a)] -> Maybe [(Agent,[([a],[a])])]
kd45 xs triples =
  if and [ maybe False (\ x -> True) b | (a,b) <- balloons ]
     then Just [ (a, maybe undefined id b) | (a,b) <- balloons ]
     else Nothing
       where rels     = [ (a, accFor a triples)  | a     <- all_agents ]
             balloons = [ (a, kd45Balloons xs r) | (a,r) <- rels       ]
```

Given a list of pairs consisting of isolated points and balloons, here is a cheap check to see whether it corresponds to a balloon list:

```
kd45psbs2balloons :: (Eq a, Ord a) =>
  [(Agent,([a],[([a],[a])]))] -> Maybe [(Agent,[([a],[a])])]
kd45psbs2balloons psbs =
  if all (\ x -> x == []) entryList
     then Just balloons
     else Nothing
  where
    entryList  = [ fst bs      | (a,bs) <- psbs ]
    balloons   = [ (a, snd bs) | (a,bs) <- psbs ]
```

Given a list of balloons, here is a cheap check to see whether it corresponds to a partition:

```
s5ball2part :: (Eq a, Ord a) =>
  [(Agent,[([a],[a])])] -> Maybe [(Agent,[[a]])]
s5ball2part balloons =
  if all (\ x -> x == []) entryList
     then Just partitions
     else Nothing
  where
    entryList  = [ concat (map fst bs) | (a,bs) <- balloons ]
    partitions = [ (a, map snd bs)     | (a,bs) <- balloons ]
```

11 Model Display

Since models can become quite large, we need a way of displaying them in a convenient fashion. The tool for this is the following display function, useful for formatting large lists of showable things:

```
display :: Show a => Int -> [a] -> IO()
display n = if n < 1 then error "parameter not positive"
                     else display' n n
  where
  display' ::  Show a => Int -> Int -> [a] -> IO()
  display' n m [] = putChar '\n'
  display' n 1 (x:xs) =   do (putStr . show) x
                             putChar '\n'
                             display' n n xs
  display' n m (x:xs) =   do (putStr . show) x
                             display' n (m-1) xs
```

Use this for displaying models, where the accessibility relations of S5 models are displayed as partitions and the accessibility relations of KD45 models as balloons (see Section A).

```
showMo :: (Eq state, Show state, Ord state, Show formula) =>
                Model state formula -> IO()
showMo = displayM 10
```

Showing Pmods:

```
showM :: (Eq state, Show state, Ord state, Show formula) =>
                               Pmod state formula -> IO()
showM (Pmod sts pre acc pnts) = do putStr "==> "
                                   print pnts
                                   showMo (Mo sts pre acc)
```

Showing lists of Pmods:

```
showMs :: (Eq state, Show state, Ord state, Show formula) =>
                               [Pmod state formula] -> IO()
showMs ms = sequence_ (map showM ms)
```

Displaying models:

```
displayM :: (Eq state, Show state, Ord state, Show formula) =>
                Int -> Model state formula -> IO()
displayM n (Mo states pre rel) =
  do print states
     display (div n 2) pre
     case (k45 states rel) of
       Nothing       -> display n rel
       Just psbs     -> case kd45psbs2balloons psbs of
         Nothing       -> displayPB (div n 2) psbs
         Just balloons -> case s5ball2part balloons of
           Nothing          ->  displayB (div n 2) balloons
           Just parts       ->  displayP (2*n) parts
```

Displaying lists of partitions:

```
displayP :: Show a => Int -> [(Agent,[[a]])] -> IO()
displayP n parts = sequence_ (map (display n) (map (\x -> [x]) parts))
```

Displaying lists of KD45 balloons:

```
displayB :: Show a => Int -> [(Agent,[([a],[a])])] -> IO()
displayB n balloons = sequence_ (map (display n) (map (\x -> [x]) balloons))
```

Displaying a list of K45 isolated points plus balloons:

```
displayPB :: Show a => Int ->  [(Agent,([a],[([a],[a])]))] -> IO()
displayPB n  psbs = sequence_ (map (display n) (map (\x -> [x]) psbs))
```

12 Model Visualisation

For graphical visualisation of static models and action models, we use the graphic visualisation tool *dot* [26]. Define a class `GraphViz`, as a subclass of `Show`.

```
class Show a => GraphViz a where
  graphviz :: a -> String
```

Every instance of the `GraphViz` class should have the function `graphviz` defined on it.

The following functions uses a string as glue between a list of strings, to produce a single long string.

```
glueWith :: String -> [String] -> String
glueWith _ []     = []
glueWith _ [y]    = y
glueWith s (y:ys) = y ++ s ++ glueWith s ys
```

Listing states:

```
listState  :: (Show a, Show b, Eq a, Eq b) => a -> [(a,b)] -> String
listState w val =
    let
      props = head (maybe [] (\ x -> [x]) (lookup w val))
      label = filter (isAlphaNum) (show props)
    in
      if null label
         then show w
         else show w
              ++ "[label =\"" ++ (show w) ++ ":" ++ (show props) ++ "\"]"
```

Get the links from a labelled relation:

```
links :: (Eq a, Eq b) => [(a,b,b)] -> [(a,b,b)]
links [] = []
links ((x,y,z):xyzs) | y == z    = links xyzs
                     | otherwise =
                       (x,y,z): links (filter (/= (x,z,y)) xyzs)
```

Compress the links from a labelled relation:

```
cmpl :: Eq b => [(Agent,b,b)] -> [([Agent],b,b)]
cmpl [] = []
cmpl ((x,y,z):xyzs) = (xs,y,z):(cmpl xyzs')
  where xs = x: [ a | a  <- all_agents, elem a (map f xyzs1) ]
        xyzs1 = filter (\ (u,v,w) ->
                       (v == y && w == z)
                               ||
                       (v == z && w == y)) xyzs
        f (x,_,_) = x
        xyzs' = xyzs \\ xyzs1
```

Put models in the class `GraphViz`, and define the `graphviz` function for them:

```
instance (Show a, Show b, Eq a, Eq b) => GraphViz (Model a b) where
  graphviz (Mo states val rel) = if isS5 states rel
   then
    "digraph G { "
      ++
      glueWith  " ; " [ listState s val | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s')
                                ++ " [label="
                                ++ (filter isAlpha (show ags))
                                ++ ",dir=none ]"  |
                         s <- states, s' <- states,
                         (ags,t,t') <- (cmpl . links) rel,
                         s == t, s' == t'                         ]
      ++ " }"
   else
    "digraph G { "
      ++
      glueWith  " ; " [ listState s val | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s')
                                ++ " [label=" ++ (show ag) ++ "]"  |
                         s <- states, s' <- states,
                         (ag,t,t') <- rel,
                         s == t, s' == t'                         ]
      ++ " }"
```

Listing pointed states:

```
listPState  :: (Show a, Show b, Eq a, Eq b) =>
                a -> [(a,b)] -> Bool -> String
listPState w val pointed =
    let
      props = head (maybe [] (\ x -> [x]) (lookup w val))
      label = filter (isAlphaNum) (show props)
    in
      if null label
         then if pointed then show w ++ "[peripheries = 2]"
              else            show w
         else if pointed then
              show w
              ++ "[label =\"" ++ (show w) ++ ":" ++ (show props) ++
                 "\",peripheries = 2]"
              else show w
              ++ "[label =\"" ++ (show w) ++ ":" ++ (show props) ++ "\"]"
```

Listing pointed models:

```
instance (Show a, Show b, Eq a, Eq b) => GraphViz (Pmod a b)  where
  graphviz (Pmod states val rel points) = if isS5 states rel
   then
    "digraph G { "
      ++
      glueWith  " ; " [ listPState s val (elem s points) | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s’)
                                ++ " [label="
                                ++ (filter isAlpha (show ags))
                                ++  ",dir=none ]"  |
                         s <- states, s’ <- states,
                         (ags,t,t’) <- (cmpl . links) rel,
                         s == t, s’ == t’                         ]
      ++ " }"
   else
    "digraph G { "
      ++
      glueWith  " ; " [ listPState s val (elem s points) | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s’)
                                ++ " [label=" ++ (show ag) ++ "]"  |
                         s <- states, s’ <- states,
                         (ag,t,t’) <- rel,
                         s == t, s’ == t’                         ]
      ++ " }"
```

Write graph to file:

```
writeGraph :: String -> IO()
writeGraph cts = writeFile "graph.dot" cts
```

```
writeGr :: String -> String -> IO()
writeGr name cts = writeFile name cts
```

Write model to file:

```
writeModel :: (Show a, Show b, Eq a, Eq b) => Model a b -> IO()
writeModel m = writeGraph (graphviz m)
```

Write Pmod to file:

```
writePmod :: (Show a, Show b, Eq a, Eq b) => (Pmod a b) -> IO()
writePmod m = writeGraph (graphviz m)
```

```
writeP :: (Show a, Show b, Eq a, Eq b) => String -> (Pmod a b) -> IO()
writeP name m = writeGr (name ++ ".dot") (graphviz m)
```

To Do 1 *Improve the display of S5, KD45 and K45 models along the lines of the terminal display code for models from the previous section.*

To Do 2 *Add visualisations of automata.*

13 Reducing Formulas to Canonical Form

For computing bisimulations, it is useful to have some notion of equivalence (however crude) for the logical language. For this, we reduce formulas to a canonical form. We will derive canonical forms that are unique up to propositional equivalence, employing a propositional reasoning engine. This is still rather crude, for any modal formula will be treated as a propositional literal.

The DPLL (Davis, Putnam, Logemann, Loveland) engine in Appendix B expects clauses represented as lists of integers, so we first have to translate to this format. This translation should start with computing a mapping from positive literals to integers.

For the non-propositional operators we use a little bootstrapping, by putting the formula inside the operator in canonical form, using the function `canonF` to be defined below. Also, since the non-propositional operators all behave as Box modalities, we can reduce $\Box\top$ to $\top$.

```
mapping :: Form -> [(Form,Integer)]
mapping f = zip lits [1..k]
  where
  lits = (sort . nub . collect) f
  k    = toInteger (length lits)
  collect :: Form -> [Form]
  collect Top         = []
  collect (Prop p)    = [Prop p]
  collect (Neg f)     = collect f
  collect (Conj fs)   = concat (map collect fs)
  collect (Disj fs)   = concat (map collect fs)
  collect (Pr pr f)   = if canonF f == Top then [] else [Pr pr (canonF f)]
  collect (K ag f)    = if canonF f == Top then [] else [K ag (canonF f)]
  collect (EK ags f)  = if canonF f == Top then [] else [EK ags (canonF f)]
  collect (CK ags f)  = if canonF f == Top then [] else [CK ags (canonF f)]
  collect (Up pam f)  = if canonF f == Top then [] else [Up pam (canonF f)]
  collect (Aut nfa f) = if nfa == nullAut || canonF f == Top
                        then [] else [Aut nfa (canonF f)]
```

Putting in clausal form, given a mapping for the literals, and using bootstrapping for formulas in the scope of a non-propositional operator. Note that $\Box\top$ is reduced to $\top$, $\neg\Box\top$ to $\bot$, and $[\text{Aut}]\varphi$ to $\top$ (and $\neg[\text{Aut}]\varphi$ to $\bot$) if Aut is the automaton for the empty language.

```
cf :: (Form -> Integer) -> Form -> [[Integer]]
cf g (Top)            = []
cf g (Prop p)         = [[g (Prop p)]]
cf g (Pr pr f)        = if canonF f == Top then []
                        else [[g (Pr pr (canonF f))]]
cf g (K ag f)         = if canonF f == Top then []
                        else [[g (K ag (canonF f))]]
cf g (EK ags f)       = if canonF f == Top then []
                        else [[g (EK ags (canonF f))]]
cf g (CK ags f)       = if canonF f == Top then []
                        else [[g (CK ags (canonF f))]]
cf g (Up am f)        = if canonF f == Top then []
                        else [[g (Up am (canonF f))]]
cf g (Aut nfa f)      = if nfa == nullAut || canonF f == Top then []
                        else [[g (Aut nfa (canonF f))]]
cf g (Conj fs)        = concat (map (cf g) fs)
cf g (Disj fs)        = deMorgan (map (cf g) fs)
```

Negated formulas:

```
cf g (Neg Top)        = [[]]
cf g (Neg (Prop p))   = [[- g (Prop p)]]
cf g (Neg (Pr pr f))  = if canonF f == Top then [[]]
                        else [[- g (Pr pr (canonF f))]]
cf g (Neg (K ag f))   = if canonF f == Top then [[]]
                        else [[- g (K ag (canonF f))]]
cf g (Neg (EK ags f)) = if canonF f == Top then [[]]
                        else [[- g (EK ags (canonF f))]]
cf g (Neg (CK ags f)) = if canonF f == Top then [[]]
                        else [[- g (CK ags (canonF f))]]
cf g (Neg (Up am f))  = if canonF f == Top then [[]]
                        else [[- g (Up am (canonF f))]]
cf g (Neg (Aut nfa f))= if nfa == nullAut || canonF f == Top then [[]]
                        else [[- g (Aut nfa (canonF f))]]
cf g (Neg (Conj fs))  = deMorgan (map (\ f -> cf g (Neg f)) fs)
cf g (Neg (Disj fs))  = concat   (map (\ f -> cf g (Neg f)) fs)
cf g (Neg (Neg f))    = cf g f
```

De Morgan's disjunction distribution:

$$\varphi \vee (\psi_1 \wedge \cdots \wedge \psi_n) \leftrightarrow (\varphi \vee \psi_1) \wedge \cdots \wedge (\varphi \wedge \psi_n).$$

De Morgan's disjunction distribution, for the case of a disjunction of a list of clause sets.

```
deMorgan :: [[[Integer]]] -> [[Integer]]
deMorgan [] = [[]]
deMorgan [cls] = cls
deMorgan (cls:clss) = deMorg cls (deMorgan clss)
  where
  deMorg :: [[Integer]] -> [[Integer]] -> [[Integer]]
  deMorg cls1 cls2 = (nub . concat) [ deM cl cls2 | cl <- cls1 ]
  deM :: [Integer] -> [[Integer]]  -> [[Integer]]
  deM cl cls = map (fuseLists cl) cls
```

Function `fuseLists` keeps the literals in the clauses ordered.

```
fuseLists :: [Integer] -> [Integer] -> [Integer]
fuseLists [] ys = ys
fuseLists xs [] = xs
fuseLists (x:xs) (y:ys) | abs x < abs y  = x:(fuseLists xs (y:ys))
                        | abs x == abs y = if x == y
                                             then x:(fuseLists xs ys)
                                             else if x > y
                                               then x:y:(fuseLists xs ys)
                                               else y:x:(fuseLists xs ys)
                        | abs x > abs y  = y:(fuseLists (x:xs) ys)
```

Given a mapping for the positive literals, the satisfying valuations of a formula can be collected from the output of the DPLL process. Here `dp` is the function imported from the module `DPLL`.

```
satVals :: [(Form,Integer)] -> Form -> [[Integer]]
satVals t f = (map fst . dp) (cf (table2fct t) f)
```

Two formulas are propositionally equivalent if they have the same sets of satisfying valuations, computed on the basis of a literal mapping for their conjunction:

```
propEquiv :: Form -> Form -> Bool
propEquiv f1 f2 = satVals g f1 == satVals g f2
  where g = mapping (Conj [f1,f2])
```

A formula is a (propositional) contradiction if it is propositionally equivalent to `Neg Top`, or equivalently, to `Disj []`:

```
contrad :: Form -> Bool
contrad f = propEquiv f (Disj [])
```

A formula is (propositionally) consistent if it is not a propositional contradiction:

```
consistent :: Form -> Bool
consistent = not . contrad
```

Use the set of satisfying valuations to derive a canonical form:

```
canonF :: Form -> Form
canonF f = if (contrad (Neg f))
             then Top
             else if fs == []
             then Neg Top
             else if length fs == 1
             then head fs
             else Disj fs
  where g   = mapping f
        nss = satVals g f
        g'  = \ i -> head [ form | (form,j) <- g, i == j ]
        h   = \ i -> if i < 0 then Neg (g' (abs i)) else g' i
        h'  = \ xs -> map h xs
        k   = \ xs -> if xs == []
                         then Top
                         else if length xs == 1
                                 then head xs
                                 else Conj xs
        fs  = map k (map h' nss)
```

This gives:

```
DEMO> canonF p
p
DEMO> canonF (Conj [p,Top])
p
DEMO> canonF (Conj [p,q,Neg r])
&[p,q,-r]
DEMO> canonF (Neg (Disj [p,(Neg p)]))
-T
DEMO> canonF (Disj [p,q,Neg r])
v[p,&[-p,q],&[-p,-q,-r]]
DEMO> canonF (K a (Disj [p,q,Neg r]))
[a]v[p,&[-p,q],&[-p,-q,-r]]
DEMO> canonF (Conj  [p, Conj [q,Neg r]])
&[p,q,-r]
DEMO> canonF (Conj  [p, Disj [q,Neg (K a (Disj []))]])
v[&[p,q],&[p,-q,-[a]-T]]
DEMO> canonF (Conj  [p, Disj [q,Neg (K a (Conj []))]])
&[p,q]
```

To Do 3 *Extend this with a further treatment of:*

- $\Box_a$ *modalities,*

- E_B *modalities,*
- C_B *modalities,*
- $[\mathbf{M}]$ *update modalities.*

Propositional decomposition in the scope of these operators is already being performed. Further decomposition is a first step. At a later stage we may consider hooking up to a proof engine for modal logic, or extending the present propositional prover with rules for the modalities.

14 Model Minimization under Bisimulation

Any Kripke model can be simplified by replacing each state s by its bisimulation class $[s]$. The problem of finding the smallest Kripke model modulo bisimulation is similar to the problem of minimizing the number of states in a finite automaton [21]. We will use partition refinement, in the spirit of [27]. Here is the algorithm:

- Start out with a partition of the state set where all states with the same precondition function are in the same class. The equality relation to be used to evaluate the precondition function is given as a parameter to the algorithm.
- Given a partition Π, for each block b in Π, partition b into sub-blocks such that two states s, t of b are in the same sub-block iff for all agents a it holds that s and t have $\xrightarrow{a}$ transitions to states in the same block of Π. Update Π to Π' by replacing each b in Π by the newly found set of sub-blocks for b.
- Halt as soon as $\Pi = \Pi'$.

Looking up and checking of two formulas against a given equivalence relation:

```
lookupFs :: (Eq a,Eq b) => a -> a -> [(a,b)] -> (b -> b -> Bool) -> Bool
lookupFs i j table r = case lookup i table of
  Nothing -> lookup j table == Nothing
  Just f1 -> case lookup j table of
      Nothing -> False
      Just f2 -> r f1 f2
```

Computing the initial partition, using a particular relation for equivalence of formulas:

```
initPartition :: (Eq a, Eq b) => Model a b -> (b -> b -> Bool) -> [[a]]
initPartition (Mo states pre rel) r =
  rel2part states (\ x y -> lookupFs x y pre r)
```

Refining a partition:

```
refinePartition :: (Eq a, Eq b) => Model a b -> [[a]] -> [[a]]
refinePartition m p = refineP m p p
  where
  refineP :: (Eq a, Eq b) => Model a b -> [[a]] -> [[a]] -> [[a]]
  refineP m part [] = []
  refineP m@(Mo states pre rel) part (block:blocks) =
     newblocks ++ (refineP m part blocks)
       where
         newblocks =
           rel2part block (\ x y -> sameAccBlocks m part x y)
```

Function that checks whether two states have the same accessible blocks under a partition:

```
sameAccBlocks :: (Eq a, Eq b) =>
         Model a b -> [[a]] -> a -> a -> Bool
sameAccBlocks m@(Mo states pre rel) part s t =
    and [ accBlocks m part s ag == accBlocks m part t  ag |
                                              ag <- all_agents ]
```

The accessible blocks for an agent from a given state, given a model and a partition:

```
accBlocks :: (Eq a, Eq b) => Model a b -> [[a]] -> a -> Agent -> [[a]]
accBlocks m@(Mo states pre rel) part s ag =
    nub [ bl part y | (ag',x,y) <- rel, ag' == ag, x == s ]
```

The block of an object in a partition:

```
bl :: (Eq a) => [[a]] -> a -> [a]
bl part x = head (filter (\ b -> elem x b) part)
```

Initializing and refining a partition:

```
initRefine :: (Eq a, Eq b) => Model a b -> (b -> b -> Bool) -> [[a]]
initRefine m r = refine m (initPartition m r)
```

The refining process:

```
refine :: (Eq a, Eq b) => Model a b -> [[a]] -> [[a]]
refine m part = if rpart == part
                       then part
                       else refine m rpart
  where rpart = refinePartition m part
```

Use this to construct the minimal model. Notice the dependence on relational parameter `r`.

```
minimalModel :: (Eq a, Ord a, Eq b, Ord b) =>
                 (b -> b -> Bool) -> Model a b -> Model [a] b
minimalModel r m@(Mo states pre rel) =
  (Mo states' pre' rel')
     where
     partition = initRefine m r
     states'   = partition
     f         = bl partition
     rel'      = (nub.sort) (map (\ (x,y,z) -> (x, f y, f z)) rel)
     pre'      = (nub.sort) (map (\ (x,y)   -> (f x, y))      pre)
```

Bisimulation-minimal Pmods:

```
minimalPmod :: (Eq a, Ord a, Eq b, Ord b) =>
                  (b -> b -> Bool) -> Pmod a b -> Pmod [a] b
minimalPmod r (Pmod sts pre rel pts) = (Pmod sts' pre' rel' pts')
  where (Mo sts' pre' rel') = minimalModel r (Mo sts pre rel)
        pts' = map (bl sts') pts
```

Converting `a`'s into integers, using their position in a given list of `a`'s.

```
convert :: (Eq a, Show a) => [a] -> a  -> Integer
convert = convrt 0
  where
  convrt :: (Eq a, Show a) => Integer -> [a] -> a -> Integer
  convrt n []     x = error (show x ++ " not in list")
  convrt n (y:ys) x | x == y    = n
                    | otherwise = convrt (n+1) ys x
```

Converting an object of type `Model a b` into an object of type `Model State b`:

```
conv ::  (Eq a, Show a) => Model a b -> Model State b
conv  (Mo worlds val acc) =
      (Mo (map f worlds)
          (map (\ (x,y)   -> (f x, y)) val)
          (map (\ (x,y,z) -> (x, f y, f z)) acc))
  where f = convert worlds
```

Conversion by renaming of Pmods:

```
convPmod ::  (Eq a, Show a) => Pmod a b -> Pmod State b
convPmod (Pmod sts pre rel pts) = (Pmod sts' pre' rel' pts')
   where (Mo sts' pre' rel') = conv (Mo sts pre rel)
         pts' = nub (map (convert sts) pts)
```

Use this to rename the blocks into integers:

```
bisim ::  (Eq a, Ord a, Show a, Eq b, Ord b) =>
           (b -> b -> Bool) -> Model a b -> Model State b
bisim r = conv . (minimalModel r)
```

Reducing Pmods under bisimulation:

```
bisimPmod ::  (Eq a, Ord a, Show a, Eq b, Ord b) =>
            (b -> b -> Bool) -> Pmod a b -> Pmod State b
bisimPmod r = convPmod . (minimalPmod r)
```

15 Model Minimization under Action Emulation

A structural relation of action emulation, weaker than bisimulation and intended to exactly capture the update effects of action models is defined in [13], as follows.

Definition 3 (Action Emulation) *If* $\mathbf{A}$ *and* $\mathbf{B}$ *are actions with sets of action states* W_A*,* W_B*, respectively, then a relation* $R \subseteq W_A \times W_B$ *is an action emulation if whenever* sRt *the following hold:*

Preconditions $pre(s) \wedge pre(t)$ *is consistent.*

Zig *If* $s \xrightarrow{a} s'$ *then there are* $t_1, \dots, t_n$ *with*

$$t \xrightarrow{a} t_1, \dots, t \xrightarrow{a} t_n, s'Rt_1, \dots, s'Rt_n \text{ and } pre(s') \models pre(t_1) \vee \cdots \vee pre(t_n).$$

Zag *If* $t \xrightarrow{a} t'$ *then there are* $s_1, \dots, s_n$ *with*

$$s \xrightarrow{a} s_1, \dots, s \xrightarrow{a} s_n, s_1Rt', \dots, s_nRt' \text{ and } pre(t') \models pre(s_1) \vee \cdots \vee pre(s_n).$$

Definition 4 *A total action emulation between* $\mathbf{A}$ *and* $\mathbf{B}$ *is an action emulation* $R \subseteq W_A \times W_B$ *satisfying the following extra requirement: For every* $s \in W_{\mathbf{A}}$ *there are* $t_1, \dots, t_n \in W_{\mathbf{B}}$ *such that* $sRt_1, \dots, sRt_n$ *and* $pre(s) \models pre(t_1) \vee \cdots \vee pre(t_n)$*, and for every* $t \in W_{\mathbf{B}}$ *there are* $s_1, \dots, s_n \in W_{\mathbf{A}}$ *with* $s_1Rt, \dots, s_nRt$ *and* $pre(t) \models pre(s_1) \vee \cdots \vee pre(s_n)$*.*

Any action model can be simplified by replacing each state s by its action emulation class $[s]$. The problem of finding the smallest action model modulo action emulation is similar to the problem of minimizing under bisimulation. We again use partition refinement [27]. Here is the adapted algorithm:

1. Take the generated submodel of the action model, while throwing away all states s with $\text{pre}(s) \equiv \bot$.
2. Use partition refinement to compute the bisimilation minimal version of the action model.
3. Do a root unfolding if the action model is multi-pointed.
4. Generate a partition Π of the new state set where all states with the same predecessors (for all agents) and the same successors (for all agents) are in the same class.
5. Use partition refinement, starting from Π, to minimize the result.
6. Set the precondition of each partition block B equal to $\bigvee_{s \in B} \text{pre}(s)$.
7. Use partition refinement again to compute the bisimulation minimal version of the new model.

This is an extension of the algorithm for finding bisimulation minimal models.

The function for doing a root unfolding.

```
unfold :: PoAM -> PoAM
unfold (Pmod states pre acc [])     = zero
unfold am@(Pmod states pre acc [p]) = am
unfold (Pmod states pre acc points) =
  Pmod states' pre' acc' points'
  where
  len = toInteger (length states)
  points' = [ p + len | p <- points ]
  states' = states ++ points'
  pre'    = pre ++ [ (j+len,f) | (j,f) <- pre, k <- points, j == k ]
  acc'    = acc ++ [ (ag,i+len,j) | (ag,i,j) <- acc, k <- points, i == k ]
```

Finding the predecessors and successors of a state, for a given relation:

```
preds, sucs :: (Eq a, Ord a, Eq b, Ord b) => [(a,b,b)] -> b -> [(a,b)]
preds rel s = (sort.nub) [ (ag,s1) | (ag,s1,s2) <- rel, s == s2 ]
sucs  rel s = (sort.nub) [ (ag,s2) | (ag,s1,s2) <- rel, s == s1 ]
```

Initializing a partition based on same predecessors and same successors:

```
psPartition :: (Eq a, Ord a, Eq b) => Model a b -> [[a]]
psPartition (Mo states pre rel) =
  rel2part states (\ x y -> preds rel x == preds rel y
                            &&
                            sucs rel x == sucs rel y)
```

Implementation of parts 4,5,6 of the algorithm for finding the action emulation minimal Pmod:

```
minPmod :: (Eq a, Ord a) => Pmod a Form -> Pmod [a] Form
minPmod pm@(Pmod states pre rel pts) =
  (Pmod states' pre' rel' pts')
     where
     m         = Mo states pre rel
     partition = refine m (psPartition m)
     states'   = partition
     f         = bl partition
     g         = \ xs -> canonF (Disj (map (table2fct pre) xs))
     rel'      = (nub.sort) (map (\ (x,y,z) -> (x, f y, f z)) rel)
     pre'      = zip states' (map g states')
     pts'      = map (bl states') pts
```

Reducing Pmods under action emulation:

```
aePmod ::  (Eq a, Ord a, Show a) => Pmod a Form -> Pmod State Form
--aePmod ::  PoAM -> PoAM
aePmod = (bisimPmod propEquiv) . minPmod . unfold .
                          (bisimPmod propEquiv) . gsmPoAM . convPmod
```

Example action models:

```
am0 = ndSum' (test p) (test (Neg p))

am1 = ndSum' (test p) (ndSum' (test q) (test r))
```

Examples of minimization for action emulation:

```
EMO> showM am0
==> [0,2]
[0,1,2,3]
(0,p)(1,T)(2,-p)(3,T)
(a,[([0],[1]),([2],[3])])
(b,[([0],[1]),([2],[3])])
(c,[([0],[1]),([2],[3])])

DEMO> showM (aePmod am0)
==> [0]
```

```
[0]
(0,T)
(a,[[0]])
(b,[[0]])
(c,[[0]])

DEMO> showM am1
==> [0,2,4]
[0,1,2,3,4,5]
(0,p)(1,T)(2,q)(3,T)(4,r)
(5,T)
(a,[([0],[1]),([2],[3]),([4],[5])])
(b,[([0],[1]),([2],[3]),([4],[5])])
(c,[([0],[1]),([2],[3]),([4],[5])])

DEMO> showM (aePmod am1)
==> [0]
[0,1]
(0,v[p,&[-p,q],&[-p,-q,r]])(1,T)
(a,[([0],[1])])
(b,[([0],[1])])
(c,[([0],[1])])
```

16 Program Transformation

For every action model A with states $s_0, \ldots, s_{n-1}$ we define a set of n^2 program transformers $T^A_{i,j}$ $(0 \leq i < n, 0 \leq j < n)$, as follows [12]:

:

$$\begin{aligned}
T^A_{ij}(a) &= \begin{cases} ?\mathrm{pre}(s_i); a & \text{if } s_i \xrightarrow{a} s_j, \\ ?\bot & \text{otherwise} \end{cases} \\
T^A_{ij}(?\varphi) &= \begin{cases} ?(\mathrm{pre}(s_i) \wedge [A, s_i]\varphi) & \text{if } i = j, \\ ?\bot & \text{otherwise} \end{cases} \\
T^A_{ij}(\pi_1; \pi_2) &= \bigcup_{k=0}^{n-1} (T^A_{ik}(\pi_1); T^A_{kj}(\pi_2)) \\
T^A_{ij}(\pi_1 \cup \pi_2) &= T^A_{ij}(\pi_1) \cup T^A_{ij}(\pi_2) \\
T^A_{ij}(\pi^*) &= K^A_{ijn}(\pi)
\end{aligned}$$

where $K^A_{ijk}(\pi)$ is a (transformed) program for all the π^* paths from s_i to s_j that can be traced through A while avoiding a pass through intermediate states s_k and higher. Thus, $K^A_{ijn}(\pi)$ is a program for all the π^* paths from s_i to s_j that can be traced through A, period.

$K^A_{ijk}(\pi)$ is defined by recursing on k, as follows:

$$K^A_{ij0}(\pi) \quad = \quad \begin{cases} ?\top \cup T^A_{ij}(\pi) & \text{if } i = j, \\ T^A_{ij}(\pi) & \text{otherwise} \end{cases}$$

$$K^A_{ij(k+1)}(\pi) \quad = \quad \begin{cases} (K^A_{kkk}(\pi))^* & \text{if } i = k = j, \\ (K^A_{kkk}(\pi))^*; K^A_{kjk}(\pi) & \text{if } i = k \neq j, \\ K^A_{ikk}(\pi); (K^A_{kkk}(\pi))^* & \text{if } i \neq k = j, \\ K^A_{ijk}(\pi) \cup (K^A_{ikk}(\pi); (K^A_{kkk}(\pi))^*; K^A_{kjk}(\pi)) & \text{otherwise } (i \neq k \neq j). \end{cases}$$

Lemma 5 (Kleene Path) *Suppose* $(w, w') \in [\![T^A_{ij}(\pi)]\!]^{\mathbf{M}}$ *iff there is a* π *path from* (w, s_i) *to* (w', s_j) *in* $\mathbf{M} \otimes A$*. Then* $(w, w') \in [\![K^A_{ijn}(\pi)]\!]^{\mathbf{M}}$ *iff there is a* π^* *path from* (w, s_i) *to* (w', s_j) *in* $\mathbf{M} \otimes A$*.*

The Kleene path lemma is the key ingredient in the proof of the following program transformation lemma.

Lemma 6 (Program Transformation) *Assume* A *has* n *states* $s_0, \ldots, s_{n-1}$*. Then:*

$$\mathbf{M} \models_w [A, s_i][\pi]\varphi \textit{ iff } \mathbf{M} \models_w \bigwedge_{j=0}^{n-1} [T^A_{ij}(\pi)][A, s_j]\varphi.$$

The implementation of the program transformation functions is given here:

```
transf :: PoAM -> Integer -> Integer -> Program -> Program
transf am@(Pmod states pre acc points) i j (Ag ag) =
   let
     f = table2fct pre i
   in
   if elem (ag,i,j) acc && f == Top          then Ag ag
   else if elem (ag,i,j) acc && f /= Neg Top then Conc [Test f, Ag ag]
   else Test (Neg Top)
transf am@(Pmod states pre acc points) i j (Ags ags) =
   let ags' = nub [ a | (a,k,m) <- acc, elem a ags, k == i, m == j ]
       ags1 = intersect ags ags'
       f    = table2fct pre i
   in
     if ags1 == [] || f == Neg Top        then Test (Neg Top)
     else if f == Top && length ags1 == 1 then Ag (head ags1)
     else if f == Top                     then Ags ags1
     else Conc [Test f, Ags ags1]
transf am@(Pmod states pre acc points) i j (Test f) =
   let
     g = table2fct pre i
   in
   if i == j
      then Test (Conj [g,(Up am f)])
      else Test (Neg Top)
transf am@(Pmod states pre acc points) i j (Conc [])  =
  transf am i j (Test Top)
transf am@(Pmod states pre acc points) i j (Conc [p]) = transf am i j p
transf am@(Pmod states pre acc points) i j (Conc (p:ps)) =
  Sum [ Conc [transf am i k p, transf am k j (Conc ps)] | k <- [0..n] ]
    where n = toInteger (length states - 1)
transf am@(Pmod states pre acc points) i j (Sum [])  =
  transf am i j (Test (Neg Top))
transf am@(Pmod states pre acc points) i j (Sum [p]) = transf am i j p
transf am@(Pmod states pre acc points) i j (Sum ps)  =
  Sum [ transf am i j p | p <- ps ]
transf am@(Pmod states pre acc points) i j (Star p) = kleene am i j n p
  where n = toInteger (length states)
```

Implementation of $K^{\mathbf{A}}_{ijk}$:

```
kleene ::  PoAM -> Integer -> Integer -> Integer -> Program -> Program
kleene am i j 0 pr =
  if i == j
    then Sum [Test Top, transf am i j pr]
    else transf am i j pr
kleene am i j k pr
  | i == j && j == pred k = Star (kleene am i i i pr)
  | i == pred k           =
    Conc [Star (kleene am i i i pr), kleene am i j i pr]
  | j == pred k           =
    Conc [kleene am i j j pr, Star (kleene am j j j pr)]
  | otherwise             =
      Sum [kleene am i j k' pr,
           Conc [kleene am i k' k' pr,
                 Star (kleene am k' k' k' pr), kleene am k' j k' pr]]
      where k' = pred k
```

Transformation plus simplification:

```
tfm ::  PoAM -> Integer -> Integer -> Program -> Program
tfm am i j pr = simpl (transf am i j pr)
```

The program transformations can be used to translate Update PDL to PDL, as follows:

$$\begin{array}{rcl}
t(\top) & = & \top \\
t(p) & = & p \\
t(\neg\varphi) & = & \neg t(\varphi) \\
t(\varphi_1 \wedge \varphi_2) & = & t(\varphi_1) \wedge t(\varphi_2) \\
t([\pi]\varphi) & = & [r(\pi)]t(\varphi) \\
t([A,s]\top) & = & \top \\
t([A,s]p) & = & t(\text{pre}(s)) \to p \\
t([A,s]\neg\varphi) & = & t(\text{pre}(s)) \to \neg t([A,s]\varphi) \\
t([A,s](\varphi_1 \wedge \varphi_2)) & = & t([A,s]\varphi_1) \wedge t([A,s]\varphi_2) \\
t([A,s_i][\pi]\varphi) & = & \bigwedge_{j=0}^{n-1} [T^A_{ij}(r(\pi))]t([A,s_j]\varphi) \\
t([A,s][A',s']\varphi) & = & t([A,s]t([A',s']\varphi)) \\
t([A,S]\varphi) & = & \bigwedge_{s\in S} t[A,s]\varphi)
\end{array}$$

$$\begin{array}{rcl}
r(a) & = & a \\
r(B) & = & B \\
r(?\varphi) & = & ?t(\varphi) \\
r(\pi_1;\pi_2) & = & r(\pi_1);r(\pi_2) \\
r(\pi_1 \cup \pi_2) & = & r(\pi_1) \cup r(\pi_2) \\
r(\pi^*) & = & (r(\pi))^*.
\end{array}$$

The correctness of this translation follows from direct semantic inspection, using the program transformation lemma for the translation of $[A,s_i][\pi]\varphi$ formulas.

The crucial clauses in this translation procedure are those for formulas of the forms $[A,S]\varphi$ and $[A,s]\varphi$, and more in particular the one for formulas of the form $[A,s][\pi]\varphi$. It makes sense to give separate functions for the steps that pull the update model through program π given formula φ.

```
step0, step1 :: PoAM -> Program -> Form -> Form
step0 am@(Pmod states pre acc []) pr f = Top
step0 am@(Pmod states pre acc [i]) pr f = step1 am pr f
step0 am@(Pmod states pre acc is) pr f =
  Conj [ step1 (Pmod states pre acc [i]) pr f | i <- is ]
step1 am@(Pmod states pre acc [i]) pr f =
   Conj [ Pr (transf am i j (rpr pr))
              (Up (Pmod states pre acc [j]) f) | j <- states ]
```

Perform a single step, and put in canonical form:

```
step :: PoAM -> Program -> Form -> Form
step am pr f = canonF (step0 am pr f)
```

```
t :: Form -> Form
t Top = Top
t (Prop p) = Prop p
t (Neg f) = Neg (t f)
t (Conj fs) = Conj (map t fs)
t (Disj fs) = Disj (map t fs)
t (Pr pr f) = Pr (rpr pr) (t f)
t (K x f)   = Pr (Ag x) (t f)
t (EK xs f)  = Pr (Ags xs) (t f)
t (CK xs f)  = Pr (Star (Ags xs)) (t f)
```

Translations of formulas starting with an action model update:

```
t (Up am@(Pmod states pre acc [i]) f) = t' am f
t (Up am@(Pmod states pre acc is) f)  =
   Conj [ t' (Pmod states pre acc [i]) f | i <- is ]
```

Translations of formulas starting with a single pointed action model update are performed by `t'`:

```
t' :: PoAM -> Form -> Form
t' am Top           = Top
t' am (Prop p)      = impl (precondition am) (Prop p)
t' am (Neg f)       =  Neg (t' am f)
t' am (Conj fs)     = Conj (map (t' am) fs)
t' am (Disj fs)     = Disj (map (t' am) fs)
t' am (K x f)       = t' am (Pr (Ag x) f)
t' am (EK xs f)     = t' am (Pr (Ags xs) f)
t' am (CK xs f)     = t' am (Pr (Star (Ags xs)) f)
t' am (Up am' f)    = t' am (t (Up am' f))
```

The crucial case: update action having scope over a program. We may assume that the update action is single pointed.

```
t' am@(Pmod states pre acc [i]) (Pr pr f) =
   Conj [ Pr (transf am i j (rpr pr))
             (t' (Pmod states pre acc [j]) f) | j <- states ]
t' am@(Pmod states pre acc is) (Pr pr f) =
   error "action model not single pointed"
```

Translations for programs:

```
rpr :: Program -> Program
rpr (Ag x)       = Ag x
rpr (Ags xs)     = Ags xs
rpr (Test f)     = Test (t f)
rpr (Conc ps)    = Conc (map rpr ps)
rpr (Sum  ps)    = Sum  (map rpr ps)
rpr (Star p)     = Star (rpr p)
```

Translating and putting in canonical form:

```
tr :: Form -> Form
tr = canonF . t
```

Some example translations:

```
DEMO> tr (Up (public p) (Pr (Star (Ags [b,c])) p))
T
DEMO> tr (Up (public (Disj [p,q])) (Pr (Star (Ags [b,c])) p))
[(U[?T,C[?v[p,q],[b,c]]])*]v[p,&[-p,-q]]
DEMO> tr (Up (groupM [a,b] p) (Pr (Star (Ags [b,c])) p))
[C[C[(U[?T,C[?p,[b,c]]])*,C[?p,[c]]],(U[U[?T,[b,c]],C[c,(U[?T,C[?p,[b,c]]])*,C[?p,[c]]]]])*]]p
DEMO> tr (Up (secret [a,b] p) (Pr (Star (Ags [b,c])) p))
[C[C[(U[?T,C[?p,[b]]])*,C[?p,[c]]],(U[U[?T,[b,c]],C[?-T,(U[?T,C[?p,[b]]])*,C[?p,[c]]]]])*]]p
```

17 Automata

Probably the translation from the previous section is all you will ever need. This section was left in for sentimental reasons.

Alphabet: accessibility steps for individual agents plus tests on formulas of the language:

```
data Symbol = Acc Agent | Tst Form deriving (Eq,Ord,Show)
```

Moves are triples consisting of start state, a symbol read and a next state.

```
data (Eq a,Ord a,Show a) => Move a = Move a Symbol a deriving (Eq,Ord,Show)
```

Automata are triples consisting of a start state, a list of possible moves and a final state (thus, the state set is left implicit).

```
data (Eq a,Ord a,Show a) => NFA a = NFA a [Move a] a deriving (Eq,Ord,Show)
```

The states of an automaton:

```
states :: (Eq a,Ord a,Show a) => NFA a -> [a]
states (NFA s delta f) = (sort . nub) (s:f:rest)
  where rest = [ s' | Move s' a t' <- delta ]
               ++
              [ t' | Move s' a t' <- delta ]
```

The symbols of an NFA:

```
symbols :: (Eq a,Ord a,Show a) => NFA a -> [Symbol]
symbols (NFA s moves f) = (sort . nub) [ symb | Move s symb t <- moves ]
```

Recognizing strings of symbols. If there is no input left, check if the start state coincides with the final state. Otherwise, construct the automata that result from reading the first symbol with the current automaton, and check if any of these accepts the rest of the input.

```
recog :: (Eq a,Ord a,Show a) => NFA a -> [Symbol] -> Bool
recog (NFA start moves final) [] = start == final
recog (NFA start moves final) (symbol:symbols) =
  any (\ aut -> recog aut symbols)
     [ NFA new moves final |
           Move s symb new <- moves, s == start, symb == symbol ]
```

Computing the reachable states of an automaton, using a well-known algorithm for graph reachability [28, Ch 1]:

```
reachable :: (Eq a,Ord a,Show a) => NFA a -> [a]
reachable (NFA start moves final) = acc moves [start] []
  where
  acc :: (Show a,Ord a) => [Move a] -> [a] -> [a] -> [a]
  acc moves [] marked = marked
  acc moves (b:bs) marked = acc moves (bs ++ (cs \\ bs)) (marked ++ cs)
     where
     cs  = nub [ c | Move b' symb c <- moves, b' == b, notElem c marked ]
```

Simplify an automaton by removing non-accessible states from the transition relation.

```
accNFA :: (Eq a,Ord a,Show a) => NFA a -> NFA a
accNFA nfa@(NFA start moves final) =
  if
    notElem final fromStart
  then
    NFA start [] final
  else
    NFA start moves' final
  where
   fromStart = reachable nfa
   moves' = [ Move x symb y | Move x symb y <- moves, elem x fromStart ]
```

Minimizing the number of states of a finite automaton can be done with the following algorithm [21]:

- Start out with a partition $\{S - \{f\}, \{f\}\}$ of the state set of the automaton (S is the set of all states, f is the final state).

- Given a partition Π, for each block b in Π, partition b into sub-blocks such that two states s, t of b are in the same sub-block iff for all symbols σ it holds that s and t have $\xrightarrow{\sigma}$ transitions to states in the same block of Π. Update Π to Π' by replacing each b in Π by the newly found set of sub-blocks for b.
- Halt as soon as $\Pi = \Pi'$.

The initial partition:

```
initPart :: (Eq a,Ord a,Show a) => NFA a -> [[a]]
initPart nfa@(NFA start moves final) = [states nfa \\ [final], [final]]
```

Refining a partition:

```
refinePart :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> [[a]]
refinePart nfa p = refineP nfa p p
  where
  refineP :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> [[a]] -> [[a]]
  refineP nfa part [] = []
  refineP nfa@(NFA start moves final) part (block:blocks) =
     newblocks ++ (refineP nfa part blocks)
       where
         newblocks =
           rel2part block (\ x y -> sameAccBl nfa part x y)
```

Function that checks whether two states have the same accessible blocks under a partition:

```
sameAccBl :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> a -> a -> Bool
sameAccBl nfa part s t =
    and [ accBl nfa part s symb == accBl nfa part t symb |
                                               symb <- symbols nfa ]
```

Accessible blocks for a symbol from a given state, given an NFA and a partition:

```
accBl :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> a -> Symbol -> [[a]]
accBl nfa@(NFA start moves final) part s symb =
   nub [ bl part y | Move x symb' y <- moves, symb' == symb, x == s ]
```

The whole algorithm:

```
compress :: (Eq a, Ord a, Show a) => NFA a -> [[a]]
compress nfa = compress' nfa (initPart nfa)
  where
  compress' :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> [[a]]
  compress' nfa part = if rpart == part
                       then part
                       else compress' nfa rpart
   where rpart = refinePart nfa part
```

Use this to construct the minimal automaton. We employ the opportunity to put the test formulas in the transition table of the automaton in canonical form.

```
minimalAut' :: (Eq a, Ord a, Show a) => NFA a -> NFA [a]
minimalAut' nfa@(NFA start moves final) = NFA start' moves' final'
  where
   (NFA st mov fin) = accNFA nfa
   partition    = compress (NFA st mov fin)
   f            = bl partition
   g (Acc ag)   = Acc ag
   g (Tst frm) = Tst (canonF frm)
   start'       = f st
   final'       = f fin
   moves'       = (nub.sort)
                     (map (\ (Move x y z) -> Move (f x) (g y) (f z)) mov)
```

Converting an automaton of type `NFA a` to one of type `NFA State`:

```
convAut :: (Eq a,Ord a,Show a) => NFA a -> NFA State
convAut aut@(NFA s delta t) =
    NFA
    (f s)
    (map (\ (Move x symb y) -> Move (f x) symb (f y)) delta)
    (f t)
    where f = convert (states aut)
```

Simplify the output of `minimalAut'`:

```
minimalAut :: (Eq a, Ord a, Show a) => NFA a -> NFA State
minimalAut = convAut . minimalAut'
```

Automaton that accepts nothing:

```
nullAut = (NFA 0 [] 1)
```

General knowledge automaton:

```
genKnown :: [Agent] -> NFA State
genKnown agents = (NFA 0 [Move 0 (Acc a) 1 | a <- agents ] 1)
```

Relativized common knowledge automaton:

```
relCknown :: [Agent] -> Form -> NFA State
relCknown agents form = (NFA 0 (Move 0 (Tst form) 1 :
                               [Move 1 (Acc a) 0 | a <- agents]) 0)
```

Common knowledge automaton:

```
cKnown :: [Agent] -> NFA State
cKnown agents = (NFA 0 [Move 0 (Acc a) 0 | a <- agents] 0)
```

Implementation of the function AUT:

```
aut' :: (Show a,Ord a) =>
            PoAM -> State -> State -> NFA a -> NFA (State,Int,a)
aut' (Pmod sts pre acc _) s t (NFA start delta final) =
  (NFA (s,0,start) delta' (t,1,final)) where
    delta' = [ Move (u,1,w) (Acc a) (v,0,x) |
                (a,u,v) <- acc,
                Move w (Acc a') x <- delta,
                 a == a' ]
               ++
             [ Move (u,0,w) (Tst (table2fct pre u)) (u,1,w) |
                 u <- sts,
                 w <- states (NFA start delta final) ]
               ++
             [ Move (u,1,v)
                (Tst (Neg (Up (Pmod sts pre acc [u])
                                    (Neg form)))) (u,1,w) |
               u <- sts,
               Move v (Tst form) w <- delta ]
```

Simplify the output of the `aut'` function:

```
aut :: (Show a,Ord a) => PoAM -> State -> State -> NFA a -> NFA State
aut am s t nfa = minimalAut (aut' am s t nfa)
```

The `aut` function is the key ingredient of the following translation function from the language of epistemic update logic to APDL:

```
tr' :: Form -> Form
tr' Top = Top
tr' (Prop p) = Prop p
tr' (Neg form) = Neg (tr' form)
tr' (Conj forms) = Conj (map tr' forms)
tr' (Disj forms) = Disj (map tr' forms)
tr' (K agent form) = K agent (tr' form)
tr' (EK agents form) = Aut (genKnown agents) (tr' form)
tr' (CK agents form) = Aut (cKnown agents) (tr' form)
```

```
tr' (Aut nfa form) = Aut (tAut nfa) (tr' form)
tr' (Up (Pmod sts pre rel []) form) = Top
tr' (Up (Pmod sts pre rel [s]) Top) = Top
tr' (Up (Pmod sts pre rel [s]) (Prop p)) =
  impl (tr' (table2fct pre s)) (Prop p)
tr' (Up (Pmod sts pre rel [s]) (Neg form)) =
  impl (tr' (table2fct pre s))
    (Neg (tr' (Up (Pmod sts pre rel [s]) form)))
tr' (Up (Pmod sts pre rel [s]) (Conj forms)) =
  Conj [ tr' (Up (Pmod sts pre rel [s]) form) | form <- forms ]
tr' (Up (Pmod sts pre rel [s]) (Disj forms)) =
  Disj [ tr' (Up (Pmod sts pre rel [s]) form) | form <- forms ]
tr' (Up (Pmod sts pre rel [s]) (K agent form)) =
  impl (tr' (table2fct pre s))
    (Conj [ K agent (tr' (Up (Pmod sts pre rel [t]) form)) |
             t <- sts ])
tr' (Up (Pmod sts pre rel [s]) (EK agents form)) =
  tr' (Up (Pmod sts pre rel [s]) (Aut (genKnown agents) form))
tr' (Up (Pmod sts pre rel [s]) (CK agents form)) =
  tr' (Up (Pmod sts pre rel [s]) (Aut (cKnown agents) form))
```

```
tr' (Up (Pmod sts pre rel [s]) (Aut nfa form)) =
  Conj [ tr' (Aut (aut (Pmod sts pre rel [s]) s t  nfa)
              (Up  (Pmod sts pre rel [t]) form)) |  t <- sts ]
tr' (Up (Pmod sts pre rel [s]) (Up (Pmod sts' pre' rel' points) form)) =
  tr' (Up (Pmod sts pre rel [s])
    (tr' (Up (Pmod sts' pre' rel' points) form)))
tr' (Up (Pmod sts pre rel points) form) =
  Conj [ tr' (Up (Pmod sts pre rel [s]) form) | s <- points ]
```

Note that this translation generalizes the translation function from [24] to the dynamic epistemic language with multiple pointed action models. The translation demonstrates that non-deterministic dynamic epistemic logic also reduces to automata PDL.

Translating and putting in canonical form:

```
kvbtr :: Form -> Form
kvbtr = canonF . tr'
```

Translation of the test formulas inside the transition relation of an automaton:

```
tAut :: NFA State -> NFA State
tAut (NFA s delta f) = NFA s (map trans delta) f
 where trans (Move u (Acc x) v)     = Move u (Acc x) v
       trans (Move u (Tst form) v) = Move u (Tst (kvbtr form)) v
```

Some example translations:

```
DEMO> kvbtr (Up (public p) (K a p))
T
DEMO> kvbtr (Up (public p) (K a q))
v[&[p,[a]v[&[p,q],-p]],-p]
DEMO> kvbtr (Up (public p) (CK [a,b] p))
T
DEMO> kvbtr (Up (public p) (CK [a,b] q))
[NFA 0 [Move 0 (Tst p) 1,Move 1 (Acc a) 0,Move 1 (Acc b) 0] 1]v[&[p,q],-p]
DEMO> tr (Up (public p) (CK [a,b] q))
[(U[?T,C[?p,[a,b]]])*]v[&[p,q],-p]
```

18 Semantics

The group alternatives of group of agents a are the states that are reachable through $\bigcup_{a \in A} R_a$.

```
groupAlts :: [(Agent,State,State)] -> [Agent] -> State -> [State]
groupAlts rel agents current =
  (nub . sort . concat) [ alternatives rel a current | a <- agents ]
```

The common knowledge alternatives of group of agents a are the states that are reachable through a finite number of R_a links, for $a \in A$.

```
commonAlts :: [(Agent,State,State)] -> [Agent] -> State -> [State]
commonAlts rel agents current =
  closure rel agents (groupAlts rel agents current)
```

The model update function takes a static model and and action model and returns an object of type `Model (State,State) [Prop]`. The up function takes an epistemic model and a PoAM and returns a Pmod. Its states are the (State,State) pairs that result from the cartesian product construction described in [2]. Note that the update function uses the truth definition (given below as isTrAt).

```
up :: EpistM -> PoAM -> Pmod (State,State) [Prop]
up  m@(Pmod worlds val acc points) am@(Pmod states pre susp actuals) =
   Pmod worlds' val' acc' points'
   where
   worlds' = [ (w,s) | w <- worlds, s <- states,
                       formula <- maybe [] (\ x -> [x]) (lookup s pre),
                       isTrAt w m formula                ]
   val'    = [ ((w,s),props) | (w,props) <- val,
                                s        <- states,
                                elem (w,s) worlds'             ]
   acc'    = [ (ag1,(w1,s1),(w2,s2)) | (ag1,w1,w2) <- acc,
                                      (ag2,s1,s2) <- susp,
                                       ag1 == ag2,
                                       elem (w1,s1) worlds',
                                       elem (w2,s2) worlds'   ]
   points' = [ (p,a) | p <- points, a <- actuals,
                       elem (p,a) worlds'                      ]
```

The appropriate notion of equivalence for the base case of the bisimulation for epistemic models is “having the same valuation”.

```
sameVal :: [Prop] -> [Prop] -> Bool
sameVal ps qs = (nub . sort) ps ==  (nub . sort) qs
```

Bisimulation minimal version of generated submodel of update result for epistemic model and PoAM:

```
upd ::  EpistM -> PoAM -> EpistM
upd sm am = (bisimPmod (sameVal) . convPmod) (up sm am)
```

Non-deterministic update with a list of PoAMs:

```
upds  :: EpistM -> [PoAM] -> EpistM
upds = foldl upd
```

For the interpretation of automata operators we need to find the worlds that are reachable from a given world in a model through a path accepted by a given automaton. This uses the following modification of the familiar graph reachability algorithm described in [28, Ch 1] (L and K are lists of pairs consisting of worlds in the model and states in the automaton):

- Start out with a list $L = [(w, s)]$, where w is the given world and s is the start state of the automaton, and with an empty list K of marked pairs.
- Repeat until L is empty:
  - Delete the first member (w', s') from L.
  - If $(s', ?\varphi, s'')$ is a move in the automaton, for some φ true at w' in the model, with $(w', s'') \notin K$, then add (w', s'') to L and to K.
  - If (s', a, s'') is a move in the automaton, for some agent label a, and $w' \xrightarrow{a} w''$ in the model, with $(w'', s'') \notin K$, then add (w'', s'') to L and to K.
- The reachable worlds are the worlds occurring in K paired with the final state of the automaton.

This algorithm is implemented by the following function:

```
reachableAut :: SM -> NFA State -> State -> [State]
reachableAut model nfa@(NFA start moves final) w =
  acc model nfa [(w,start)] []
  where
    acc :: SM -> NFA State -> [(State,State)] -> [(State,State)] -> [State]
    acc model (NFA start moves final) [] marked =
       (sort.nub) (map fst (filter (\ x -> snd x == final) marked))
    acc m@(Mo states _ rel) nfa@(NFA start moves final)
          ((w,s):pairs) marked =
      acc m nfa (pairs ++ (cs \\ pairs)) (marked ++ cs)
      where
        cs = nub ([ (w, s') | Move t (Tst f) s' <- moves,
                              t == s, notElem (w,s') marked,
                              isTrueAt w m f ]
                   ++
                  [ (w',s') | Move t (Acc ag) s' <- moves, t == s,
                            w' <- states,
                            notElem (w',s') marked,
                            elem (ag,w,w') rel ])
```

At last we have all ingredients for the truth definition.

```
isTrueAt :: State -> SM -> Form -> Bool
isTrueAt w m Top = True
isTrueAt w m@(Mo worlds val acc) (Prop p) =
  elem p (concat [ props | (w',props) <- val, w'==w ])
isTrueAt w m (Neg f)   = not (isTrueAt w m f)
isTrueAt w m (Conj fs) = and (map (isTrueAt w m) fs)
isTrueAt w m (Disj fs) = or  (map (isTrueAt w m) fs)
```

The clauses for individual knowledge, general knowledge and common knowledge use the functions `alternatives`, `groupAlts` and `commonAlts` to compute the relevant accessible worlds:

```
isTrueAt w m@(Mo worlds val acc) (K ag f) =
  and (map (flip ((flip isTrueAt) m) f) (alternatives acc ag w))
isTrueAt w m@(Mo worlds val acc) (EK agents f) =
  and (map (flip ((flip isTrueAt) m) f) (groupAlts acc agents w))
isTrueAt w m@(Mo worlds val acc) (CK agents f) =
  and (map (flip ((flip isTrueAt) m) f) (commonAlts acc agents w))
```

In the clause for $[\mathbf{M}]\varphi$, the result of updating the static model M with action model $\mathbf{M}$ may be undefined, but in this case the precondition $P(s_0)$ of the designated state s_0 of $\mathbf{M}$ will fail in the designated world w_0 of M. By making the clause for $[\mathbf{M}]\varphi$ check for $M \models_{w_0} P(s_0)$, truth can be defined as a total function.

```
isTrueAt w m (Up am f) =
  and [ isTrueAt w' m' f |
         (m',w') <- decompose (upd (mod2pmod m [w]) am) ]
isTrueAt w m (Aut nfa f) =
  and [ isTrueAt w' m f | w' <- reachableAut m nfa w ]
```

Checking for truth in the actual world of an epistemic model:

```
isTrAt :: State -> EpistM -> Form -> Bool
isTrAt w (Pmod worlds val rel pts) = isTrueAt w (Mo worlds val rel)
```

Checking for truth in *all* the designated points of an epistemic model:

```
isTrue :: EpistM -> Form -> Bool
isTrue (Pmod worlds val rel pts) form =
   and [ isTrueAt w (Mo worlds val rel) form | w <- pts ]
```

19 Tools for Constructing Epistemic Models

The following function constructs an initial epistemic model where the agents are completely ignorant about their situation, as described by a list of basic propositions. The input is a list of basic propositions used for constructing the valuations.

```
initE :: [Prop] -> EpistM
initE allProps = (Pmod worlds val accs points)
  where
    worlds = [0..(2^k - 1)]
    k      = length allProps
    val    = zip worlds (sortL (powerList allProps))
    accs   = [ (ag,st1,st2) | ag <- all_agents,
                              st1 <- worlds,
                              st2 <- worlds      ]
    points = worlds
```

This uses the following utilities:

```
powerList  :: [a] -> [[a]]
powerList  [] = [[]]
powerList  (x:xs) = (powerList xs) ++ (map (x:) (powerList xs))

sortL :: Ord a => [[a]] -> [[a]]
sortL  = sortBy (\ xs ys -> if length xs < length ys then LT
                            else if length xs > length ys then GT
                            else compare xs ys)
```

Some initial models:

```
e00 :: EpistM
e00 = initE [P 0]

e0 :: EpistM
e0 = initE [P 0,Q 0]
```

20 From Communicative Actions to Action Models

Computing the update for a public announcement:

```
public :: Form -> PoAM
public form =
   (Pmod [0] [(0,form)] [ (a,0,0) | a <- all_agents ] [0])
```

Public announcements are S5 models:

```
DEMO> showM (public p)
==> [0]
[0]
(0,p)
(a,[[0]])
(b,[[0]])
(c,[[0]])
```

Computing the update for passing a group announcement to a list of agents: the other agents may or may not be aware of what is going on. In the limit case where the message is passed to all agents, the message is a public announcement.

```
groupM :: [Agent] -> Form -> PoAM
groupM agents form =
  if (sort agents) == all_agents
    then public form
    else
      (Pmod
         [0,1]
         [(0,form),(1,Top)]
         ([ (a,0,0) | a <- all_agents ]
           ++ [ (a,0,1) | a <- all_agents \\ agents ]
           ++ [ (a,1,0) | a <- all_agents \\ agents ]
           ++ [ (a,1,1) | a <- all_agents           ])
         [0])
```

Group announcements are S5 models:

```
DEMO> showM (groupM [a,b] p)
==> [0]
[0,1]
(0,p)(1,T)
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
```

Computing the update for an individual message to b that φ:

```
message :: Agent -> Form -> PoAM
message agent form = groupM [agent] form
```

Computing the update for passing a *secret* message to a list of agents: the other agents remain unaware of the fact that something goes on. In the limit case where the secret is divulged to all agents, the secret becomes a public update.

```
secret :: [Agent] -> Form -> PoAM
secret agents form =
  if (sort agents) == all_agents
    then public form
    else
      (Pmod
         [0,1]
         [(0,form),(1,Top)]
         ([ (a,0,0) | a <- agents ]
           ++ [ (a,0,1) | a <- all_agents \\ agents ]
           ++ [ (a,1,1) | a <- all_agents           ])
         [0])
```

Secret messages are KD45 models:

```
DEMO> showM (secret [a,b] p)
==> [0]
[0,1]
(0,p)(1,T)
(a,[([],[0]),([],[1])])
(b,[([],[0]),([],[1])])
(c,[([0],[1])])
```

To Do 4 *Add functions for messages with bcc.*

A special case of a secret is a test. Tests are updates that are kept secret from all agents:

```
test :: Form -> PoAM
test = secret []
```

Testing for $p \vee q$ is done with the following KD45 action model:

```
DEMO> showM (test (Disj [p,q]))
==> [0]
[0,1]
(0,v[p,q])(1,T)
(a,[([0],[1])])
(b,[([0],[1])])
(c,[([0],[1])])
```

Updating a model (M, w_0) with a test φ? yields a unit list containing (M, w_0) in case $M \models_{w_0} \varphi$, the empty list otherwise:

```
DEMO> showMs (upd (initM [P 0, Q 0] [P 0]) (test p))
==> 1
[0,1,2,3]
(0,[])(1,[p])(2,[q])(3,[p,q])
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
(c,[[0,1,2,3]])

DEMO> showMs (upd (initM [P 0, Q 0] [P 0]) (test (Neg p)))
```

Here is a multiple pointed action model for the communicative action of revealing one of a number of alternatives to a list of agents, in such a way that it is common knowledge that one of the alternatives gets revealed (in [3] this is called *common knowledge of alternatives*).

```
reveal :: [Agent] -> [Form] -> PoAM
reveal ags forms =
  (Pmod
     states
     (zip states forms)
     ([ (ag,s,s) | s <- states, ag <- ags ]
       ++
      [ (ag,s,s') | s <- states, s' <- states, ag <- others ])
     states)
    where states = map fst (zip [0..] forms)
          others = all_agents \\ ags
```

Here is an action model for the communication that reveals to a one of p_1, q_1, r_1.

```
DEMO> showM (reveal [a] [p1,q1,r1])
==> [0,1,2]
[0,1,2]
(0,p1)(1,q1)(2,r1)
(a,[[0],[1],[2]])
(b,[[0,1,2]])
(c,[[0,1,2]])
```

The negation of a formula:

```
negation :: Form -> Form
negation (Neg form) = form
negation form       = Neg form
```

A group of agents B gets (transparantly) informed about a formula φ if B get to know φ when φ is true, and B get to know the negation of φ otherwise. Transparancy means that all other agents are aware of the fact that B get informed about φ, i.e., the other agents learn that $(\varphi \to C_B\varphi) \wedge (\neg\varphi \to C_B\neg\varphi)$. This action model can be defined in terms of `reveal`, as follows:

```
info :: [Agent] -> Form -> PoAM
info agents form = reveal agents [form, negation form]
```

An example application:

```
DEMO> showMs (upd (initM [P 0, Q 0] [P 0]) (info [a,b] q))
==> 1
[0,1,2,3]
(0,[])(1,[p])(2,[q])(3,[p,q])
(a,[[0,1],[2,3]])
(b,[[0,1],[2,3]])
(c,[[0,1,2,3]])

DEMO> isTrue (head (upd (initM [P 0, Q 0] [P 0]) (info [a,b] q))) (CK [a,b] (Neg q))
True
```

21 Operations on Action Models

The trivial update action model is a special case of public announcement. Call this the `one` action model, for it behaves as 1 for the operation $\otimes$ of action model composition.

```
one :: PoAM
one = public Top
```

Composition $\otimes$ of multiple pointed action models.

```
cmpP :: PoAM -> PoAM ->
                 Pmod (State,State) Form
cmpP m@(Pmod states pre susp ss) (Pmod states' pre' susp' ss') =
  (Pmod nstates npre nsusp npoints)
       where
         npoints = [ (s,s') | s <- ss, s' <- ss' ]
         nstates = [ (s,s') | s <- states, s' <- states' ]
         npre    = [ ((s,s'), g) | (s,f)     <- pre,
                                   (s',f')   <- pre',
                                    g        <- [computePre m f f']      ]
         nsusp   = [ (ag,(s1,s1'),(s2,s2')) | (ag,s1,s2)    <- susp,
                                              (ag',s1',s2') <- susp',
                                               ag == ag'               ]
```

Utility function for this: compute the new precondition of a state pair. If the preconditions of the two states are purely propositional, we know that the updates at the states commute and that their combined precondition is the conjunction of the two preconditions, provided this conjunction is not a contradiction. If one of the states has a precondition that is not purely propositional, we have to take the epistemic effect of the update into account in the new precondition.

```
computePre  :: PoAM -> Form -> Form -> Form
computePre m g g'  | pureProp conj = conj
                   | otherwise     = Conj [ g, Neg (Up m (Neg g')) ]
  where conj    = canonF (Conj [g,g'])
```

To Do 5 *Refine the precondition computation, by making more clever use of what is known about the update effect of the first action model.*

Compose pairs of multiple pointed action models, and reduce the result to its simplest possible form under action emulation.

```
cmpPoAM :: PoAM -> PoAM -> PoAM
cmpPoAM pm pm' = aePmod (cmpP pm pm')
```

Use `one` as unit for composing lists of PoAMs:

```
cmp :: [PoAM] -> PoAM
cmp = foldl cmpPoAM one
```

Here is the result of composing two messages:

```
DEMO> showM (cmp [groupM [a,b] p, groupM [b,c] q])
==> [0]
[0,1,2,3]
(0,&[p,q])(1,p)(2,q)(3,T)
(a,[[0,1],[2,3]])
(b,[[0],[1],[2],[3]])
(c,[[0,2],[1,3]])
```

This gives the resulting action model. Here is the result of composing the messages in the reverse order:

```
DEMO> showM (cmp [groupM [b,c] q, groupM [a,b] p])
==> [0]
[0,1,2,3]
(0,&[p,q])(1,q)(2,p)(3,T)
(a,[[0,2],[1,3]])
(b,[[0],[1],[2],[3]])
(c,[[0,1],[2,3]])
```

These two action models are bisimilar under the renaming $1 \mapsto 2, 2 \mapsto 1$.

Here is an illustration of an observation from [11].

```
m2  = initE [P 0,Q 0]
psi = Disj[Neg(K b p),q]
```

```
DEMO> showM (upds m2 [message a psi, message b p])
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
(a,[[0,1,2,3,4,5]])
(b,[[0,2,3,5],[1,4]])
(c,[[0,1,2,3,4,5]])
DEMO> showM (upds m2 [message b p, message a psi])
==> [7]
[0,1,2,3,4,5,6,7,8,9,10]
```

```
(0,[])(1,[])(2,[p])(3,[p])(4,[p])
(5,[q])(6,[q])(7,[p,q])(8,[p,q])(9,[p,q])
(10,[p,q])
(a,[[0,3,5,7,9],[1,2,4,6,8,10]])
(b,[[0,1,3,4,5,6,9,10],[2,7,8]])
(c,[[0,1,2,3,4,5,6,7,8,9,10]])
```

Power of action models:

```
pow :: Int -> PoAM -> PoAM
pow n am = cmp (take n (repeat am))
```

The Van Benthem test: keep updating an epistemic model with the same action model until a fixpoint is reached.

```
vBtest :: EpistM -> PoAM -> [EpistM]
vBtest m a = map (upd m) (star one cmpPoAM a)

star :: a -> (a -> a -> a) -> a -> [a]
star z f a = z : star (f z a) f a
```

Putting in the fixpoint:

```
vBfix :: EpistM -> PoAM -> [EpistM]
vBfix m a = fix (vBtest m a)

fix :: Eq a => [a] -> [a]
fix (x:y:zs) = if x == y then [x]
                         else x: fix (y:zs)
```

Illustration of the above for the update with an action model **S** based on

$$p_1 \wedge (\Box_a p_1 \Rightarrow p_2) \wedge (\Box_a p_2 \Rightarrow p_3).$$

```
m1  = initE [P 1,P 2,P 3]
phi = Conj[p1,Neg (Conj[K a p1,Neg p2]),
              Neg (Conj[K a p2,Neg p3])]
a1  = message a phi
```

Three updates with $A =$ `a1` are needed before a is aware of the three facts p_1, p_2, p_3, and four before the update process reaches its fixpoint. The example shows that $A \not\leftrightarrow A^2 \not\leftrightarrow A^3 \not\leftrightarrow A^4 \leftrightarrow A^5$. Note that it takes a *long* time to generate these five updates.

```
DEMO> showMs (vBtest m1 a1)
==> [0,1,2,3,4,5,6,7]
[0,1,2,3,4,5,6,7]
(0,[])(1,[p1])(2,[p2])(3,[p3])(4,[p1,p2])
(5,[p1,p3])(6,[p2,p3])(7,[p1,p2,p3])
(a,[[0,1,2,3,4,5,6,7]])
(b,[[0,1,2,3,4,5,6,7]])
(c,[[0,1,2,3,4,5,6,7]])
==> [1,5,7,10]
[0,1,2,3,4,5,6,7,8,9,10,11]
(0,[])(1,[p1])(2,[p1])(3,[p2])(4,[p3])
(5,[p1,p2])(6,[p1,p2])(7,[p1,p3])(8,[p1,p3])(9,[p2,p3])
(10,[p1,p2,p3])(11,[p1,p2,p3])
(a,[[0,2,3,4,6,8,9,11],[1,5,7,10]])
(b,[[0,1,2,3,4,5,6,7,8,9,10,11]])
(c,[[0,1,2,3,4,5,6,7,8,9,10,11]])
==> [5,11]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13]
(0,[])(1,[p1])(2,[p1])(3,[p2])(4,[p3])
(5,[p1,p2])(6,[p1,p2])(7,[p1,p2])(8,[p1,p3])(9,[p1,p3])
(10,[p2,p3])(11,[p1,p2,p3])(12,[p1,p2,p3])(13,[p1,p2,p3])
(a,[[0,2,3,4,7,9,10,13],[1,6,8,12],[5,11]])
(b,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13]])
(c,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13]])
==> [8]
[0,1,2,3,4,5,6,7,8,9,10]
(0,[])(1,[p1])(2,[p2])(3,[p3])(4,[p1,p2])
(5,[p1,p2])(6,[p1,p3])(7,[p2,p3])(8,[p1,p2,p3])(9,[p1,p2,p3])
(10,[p1,p2,p3])
(a,[[0,1,2,3,5,6,7,10],[4,9],[8]])
(b,[[0,1,2,3,4,5,6,7,8,9,10]])
(c,[[0,1,2,3,4,5,6,7,8,9,10]])
==> [7]
[0,1,2,3,4,5,6,7,8]
(0,[])(1,[p1])(2,[p2])(3,[p3])(4,[p1,p2])
(5,[p1,p3])(6,[p2,p3])(7,[p1,p2,p3])(8,[p1,p2,p3])
(a,[[0,1,2,3,4,5,6,8],[7]])
(b,[[0,1,2,3,4,5,6,7,8]])
(c,[[0,1,2,3,4,5,6,7,8]])
==> [7]
[0,1,2,3,4,5,6,7,8]
(0,[])(1,[p1])(2,[p2])(3,[p3])(4,[p1,p2])
(5,[p1,p3])(6,[p2,p3])(7,[p1,p2,p3])(8,[p1,p2,p3])
(a,[[0,1,2,3,4,5,6,8],[7]])
(b,[[0,1,2,3,4,5,6,7,8]])
(c,[[0,1,2,3,4,5,6,7,8]])
```

Non-deterministic sum $\oplus$ of multiple-pointed action models:

```
ndSum' :: PoAM -> PoAM -> PoAM
ndSum' m1 m2 = (Pmod states val acc ss)
  where
       (Pmod states1 val1 acc1 ss1) = convPmod m1
       (Pmod states2 val2 acc2 ss2) = convPmod m2
       f   = \ x -> toInteger (length states1) + x
       states2' = map f states2
       val2'    = map (\ (x,y)   -> (f x, y)) val2
       acc2'    = map (\ (x,y,z) -> (x, f y, f z)) acc2
       ss       = ss1 ++ map f ss2
       states   = states1 ++ states2'
       val      = val1 ++ val2'
       acc      = acc1 ++ acc2'
```

Non-deterministic sum $\oplus$ of multiple-pointed action models, reduced for action emulation:

```
ndSum :: PoAM -> PoAM -> PoAM
ndSum m1 m2 = aePmod (ndSum' m1 m2)
```

Notice the difference with the definition of alternative composition of Kripke models for processes given in [20, Ch 4].

The `zero` action model is the 0 for the $\oplus$ operation, so it can be used as the base case in the following list version of the $\oplus$ operation:

```
zero :: PoAM
zero = Pmod [] [] [] []

ndS :: [PoAM] -> PoAM
ndS = foldl ndSum zero
```

Performing a test whether φ and announcing the result:

```
testAnnounce :: Form -> PoAM
testAnnounce form = ndS [ cmp [ test form, public form ],
                          cmp [ test (negation form),
                                public (negation form)] ]
```

`testAnnounce form` is equivalent to `info all_agents form`:

```
DEMO> showM (testAnnounce p)
==> [0,1]
[0,1]
(0,p)(1,-p)
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0],[1]])

DEMO> showM (info all_agents p)
==> [0,1]
[0,1]
(0,p)(1,-p)
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0],[1]])
```

The function `testAnnounce` gives the special case of revelations where the alternatives are a formula and its negation, and where the result is publicly announced.

Note that *DEMO* correctly computes the result of the sequence and the sum of two contradictory propositional tests:

```
DEMO> showM (cmp [test p, test (Neg p)])
==> []
[]

(a,[])
(b,[])
(c,[])

DEMO> showM (ndS [test p, test (Neg p)])
==> [0]
[0]
(0,T)
(a,[[0]])
(b,[[0]])
(c,[[0]])
```

22 Example: Muddy Children

Three children, a, b, c. Use p for "a is dirty", $\neg p$ for "a is clean", q, r for the same statements about b and c, respectively.

Here are action models expressing the following facts:

- a, c are aware of what is visible on b's head (aware whether q),
- a, b are aware of what is visible on c's head (aware whether r),
- b, c are aware of what is visible on a's head (aware whether p).

```
revealac_q = info [a,c] q
revealab_r = info [a,b] r
revealbc_p = info [b,c] p
```

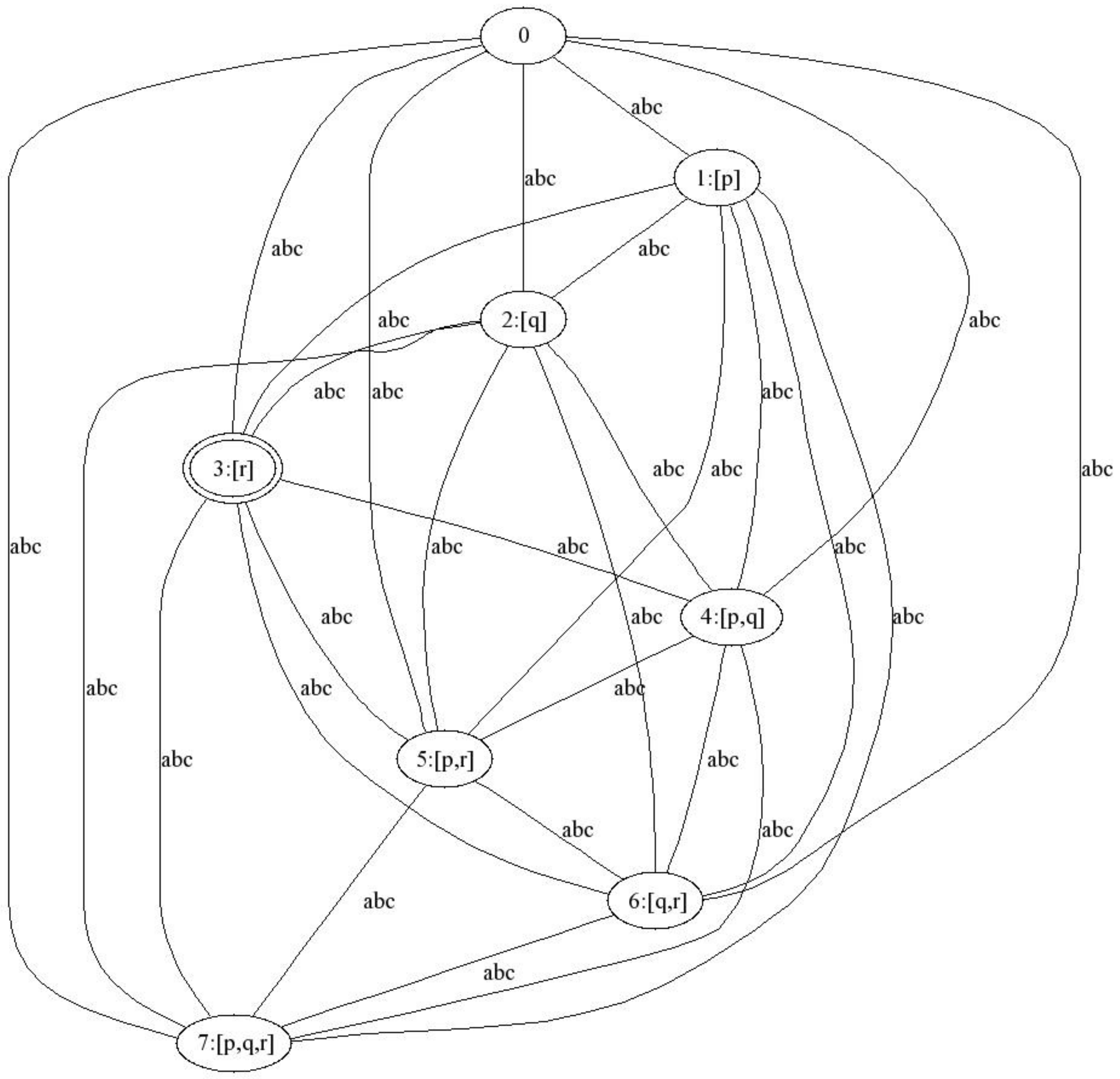
Suppose we start out with a state of affairs where everyone is completely ignorant about the facts, while in fact c is the only child that is dirty. This state of affairs is represented by:

```
initMuddy = upd (initE [P 0, Q 0, R 0]) (test (Conj [Neg p,Neg q, r]))
```

Here it is displayed:

```
DEMO> showM initMuddy
==> [3]
[0,1,2,3,4,5,6,7]
(0,[])(1,[p])(2,[q])(3,[r])(4,[p,q])
(5,[p,r])(6,[q,r])(7,[p,q,r])
(a,[[0,1,2,3,4,5,6,7]])
(b,[[0,1,2,3,4,5,6,7]])
(c,[[0,1,2,3,4,5,6,7]])
```

A different representation can be generated with `writeP "initMuddy" initMuddy`:



Updating with the awareness of the children of what they see on the other children's foreheads:

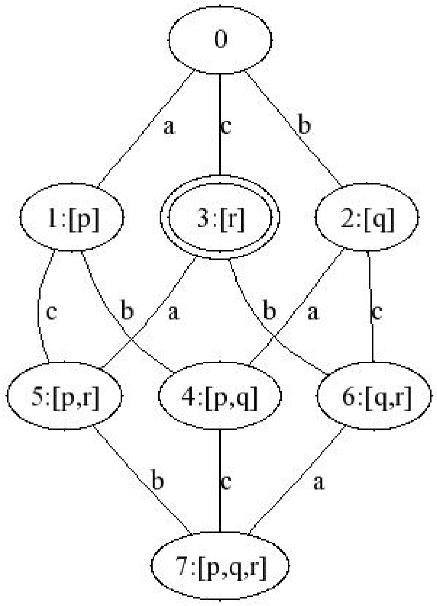
```
initMud = upds initMuddy [revealac_q,revealab_r,revealbc_p]
```

This is what the model looks like now:

```
DEMO> showM initMud
==> [3]
[0,1,2,3,4,5,6,7]
(0,[])(1,[p])(2,[q])(3,[r])(4,[p,q])
(5,[p,r])(6,[q,r])(7,[p,q,r])
```

```
(a,[[0,1],[2,4],[3,5],[6,7]])
(b,[[0,2],[1,4],[3,6],[5,7]])
(c,[[0,3],[1,5],[2,6],[4,7]])
```

This is what the model looks like under a different guise:



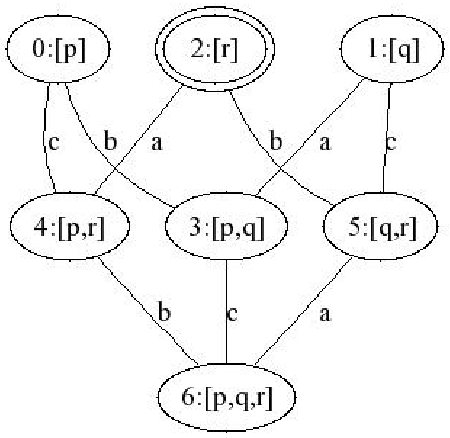
The result of updating with the public announcement of the statement that at least one child is dirty:

```
atleast1d = Disj[p,q,r]

round0 = upd initMud (public atleast1d)
```

This is what it looks like:

```
DEMO> showM round0
==> [2]
[0,1,2,3,4,5,6]
(0,[p])(1,[q])(2,[r])(3,[p,q])(4,[p,r])
(5,[q,r])(6,[p,q,r])
(a,[[0],[1,3],[2,4],[5,6]])
(b,[[0,3],[1],[2,5],[4,6]])
(c,[[0,4],[1,5],[2],[3,6]])
```

What it looks like under a different guise:



The public announcement of c that she knows whether she is dirty, and the result of updating with this:

```
cKnows = Disj [K c r, K c (Neg r)]

round1 = upd round0 (public cKnows)
```

We get:

```
DEMO> showM round1
==> [0]
[0]
(0,[r])
(a,[[0]])
(b,[[0]])
(c,[[0]])
```

Let us redo this for the case where there are two dirty children instead of one. The following gives us an initial model where a is clean and b, c are dirty.

```
initMuddy1 = upd (initE [P 0, Q 0, R 0]) (test (Conj [Neg p, q, r]))
```

Updating with the awareness of the children of what they see on the other children's foreheads:

```
initMud1 = upds initMuddy1 [revealac_q,revealab_r,revealbc_p]
```

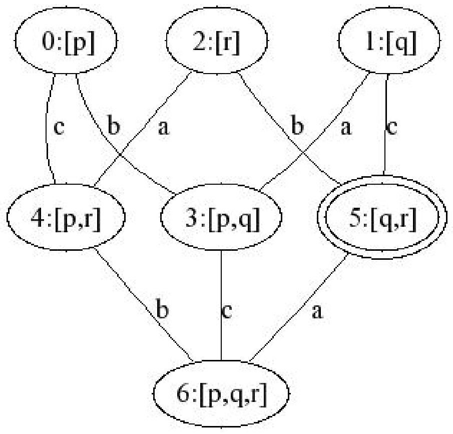
Updating with the information that at least one child is muddy:

```
nround0 = upd initMud1 (public atleast1d)
```

This is what the model looks like after this initialisation:

```
DEMO> showM nround0
==> [5]
[0,1,2,3,4,5,6]
(0,[p])(1,[q])(2,[r])(3,[p,q])(4,[p,r])
(5,[q,r])(6,[p,q,r])
(a,[[0],[1,3],[2,4],[5,6]])
(b,[[0,3],[1],[2,5],[4,6]])
(c,[[0,4],[1,5],[2],[3,6]])
```

Under a different guise:



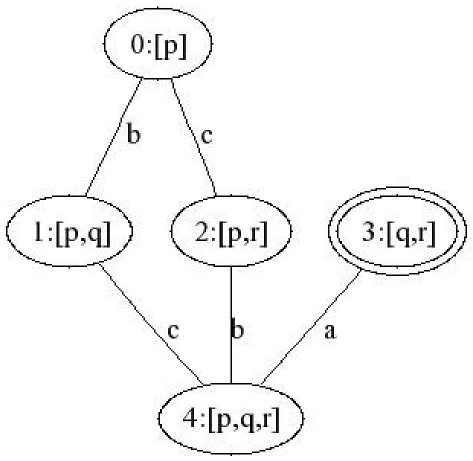
In the first round, *b* and *c* announce that they do not know their states:

```
bcKnowNot = Conj [Neg (K b q), Neg (K b (Neg q)),
                  Neg (K c r), Neg (K c (Neg r))]

nround1 = upd nround0 (public bcKnowNot)
```

We get:

```
DEMO> showM nround1
==> [3]
[0,1,2,3,4]
(0,[p])(1,[p,q])(2,[p,r])(3,[q,r])(4,[p,q,r])

(a,[[0],[1],[2],[3,4]])
(b,[[0,1],[2,4],[3]])
(c,[[0,2],[1,4],[3]])
```

Different guise:



In the second round, *c* announces that she does know her state:

```
nround2 = upd nround1 (public cKnows)
```

The result:

```
DEMO> showM nround2
==> [0]
[0]
(0,[q,r])
(a,[[0]])
(b,[[0]])
(c,[[0]])
```

23 The Measure of Ignorance

Looking at some update examples examples, we see the size of the models, measured in terms of number of transitions in the labelled transition component, may increase as we model a succession of group updates:

```
DEMO> length (access (fst (pmod2mp initMuddy)))
192
DEMO> length (access (fst (pmod2mp (upds initMuddy [groupM [a,b] r]))))
304
DEMO> length (access (fst (pmod2mp (upds initMuddy [groupM [a,b] r, groupM [b,c] (Neg p)]))))
460
```

What this shows is that 'number of transitions' is *not* a good measure for the size of an epistemic model.

For KD45 (and hence S5) models the following very simple alternative is much more attractive. For each agent, use the

> number of worlds still to be eliminated from the balloon pointed at by the actual world of the model or from the partition block containing the actual world of the model

as a measure of ignorance for that agent. This is implemented as follows:

```
measure :: (Eq a,Ord a) => (Model a b,a) -> Maybe [Int]
measure (m,w) =
  let
    f           = filter (\ (us,vs) -> elem w us || elem w vs)
    g [(xs,ys)] = length ys - 1
  in
    case kd45 (domain m) (access m) of
      Just a_balloons -> Just
         ( [ g (f balloons) | (a,balloons) <- a_balloons  ])
      Nothing         -> Nothing
```

We now get that the measure of ignorance of a model decreases with each epistemic update:

```
DEMO> map measure  (decompose initMuddy)
[Just [7,7,7]]
DEMO> map measure  (decompose (upds initMuddy [groupM [a,c] (Neg q)]))
[Just [3,11,3]]
DEMO> map measure  (decompose (upds initMuddy [groupM [a,c] (Neg q), groupM [a,b] r]))
[Just [1,5,5]]
```

This may still be too crude.

To Do 6 *Refine the measure by distinguishing between factual measure, measure up to epistemic depth* 1*, and so on. The factual measure should count the worlds modulo "having the same valuation" (making the same factual statements true), the measure up to depth* 1 *should count the worlds modulo "being bisimilar up to depth* 1 *(making the same formulas with epistemic depth up to* 1 *true), and so on. How do the results compare with the implemented version of 'measure'?*

To Do 7 *Investigate the general situation. Call an update on S5 or KD45 models honest if it does not increase the measure. What are the honest updates? Can we prove that these are precisely the updates that do not involve lying?*

24 Example: Card Showing

A simple card showing situation goes as follows.[3] Alice, Bob and Carol each hold one of cards Purple, Qaki (Khaki), Red. The actual deal is: Alice holds Purple, Bob holds Qaki, Carol holds Red. The actual action is: Alice shows Purple to Bob with Carol looking on.

The initial state of the game:

```
cards0 :: EpistM
cards0 = (Pmod [0..5] val acc [0])
  where
  val    = [(0,[P 1,Q 2,R 3]),(1,[P 1,R 2,Q 3]),
            (2,[Q 1,P 2,R 3]),(3,[Q 1,R 2,P 3]),
            (4,[R 1,P 2,Q 3]),(5,[R 1,Q 2,P 3])]
  acc    = [(a,0,0),(a,0,1),(a,1,0),(a,1,1),
            (a,2,2),(a,2,3),(a,3,2),(a,3,3),
            (a,4,4),(a,4,5),(a,5,4),(a,5,5),
            (b,0,0),(b,0,5),(b,5,0),(b,5,5),
            (b,2,2),(b,2,4),(b,4,2),(b,4,4),
            (b,3,3),(b,3,1),(b,1,3),(b,1,1),
            (c,0,0),(c,0,2),(c,2,0),(c,2,2),
            (c,3,3),(c,3,5),(c,5,3),(c,5,5),
            (c,4,4),(c,4,1),(c,1,4),(c,1,1)]
```

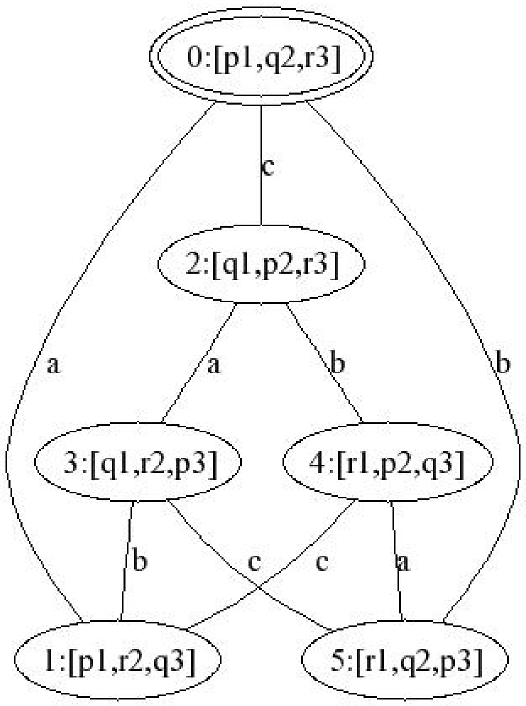
Here it is displayed:

```
DEMO> showM cards0
==> 0
```

[3]With thanks to Hans van Ditmarsch.

```
[0,1,2,3,4,5]
(0,[p1,q2,r3])(1,[p1,r2,q3])(2,[q1,p2,r3])(3,[q1,r2,p3])(4,[r1,p2,q3])
(5,[r1,q2,p3])
(a,[[0,1],[2,3],[4,5]])
(b,[[0,5],[1,3],[2,4]])
(c,[[0,2],[1,4],[3,5]])
```

Viewed as a graph:



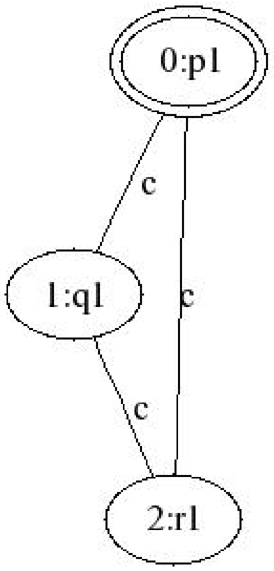
Action: *a* shows *p* to *b* with *c* looking on (*c* sees that a card is shown, but does not see that it is *p*):

```
showABp :: PoAM
showABp = (Pmod [0,1,2] pre susp [0])
  where
  pre  = [(0,p1),(1,q1),(2,r1)]
  susp = [(a,0,0),(a,1,1),(a,2,2),
          (b,0,0),(b,1,1),(b,2,2),
          (c,0,0),(c,0,1),(c,0,2),
          (c,1,0),(c,1,1),(c,1,2),
          (c,2,0),(c,2,1),(c,2,2)]
```

This gives:

```
DEMO> showM showABp
==> [0]
[0,1,2]
(0,p1)(1,q1)(2,r1)
(a,[[0],[1],[2]])
(b,[[0],[1],[2]])
(c,[[0,1,2]])
```

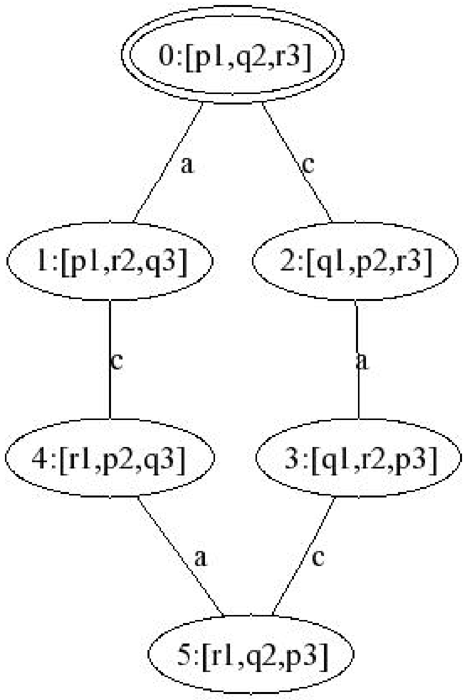
As a graph:



The result of updating with this:

```
DEMO> showM (upd cards0 showABp)
==> [0]
[0,1,2,3,4,5]
(0,[p1,q2,r3])(1,[p1,r2,q3])(2,[q1,p2,r3])(3,[q1,r2,p3])(4,[r1,p2,q3])
(5,[r1,q2,p3])
(a,[[0,1],[2,3],[4,5]])
(b,[[0],[1],[2],[3],[4],[5]])
(c,[[0,2],[1,4],[3,5]])
```

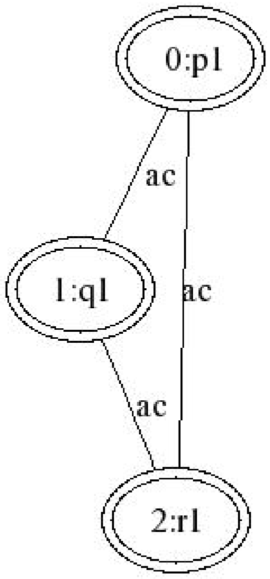
Viewed as a graph:



This modelling of the show action is still rather crude, for it is hard-coded in the action model that p_1 holds in the actual world. We can do better if we use a multiple pointed reveal action model.

```
revealAB = reveal [b] [p1,q1,r1]
```

Viewed as a graph:

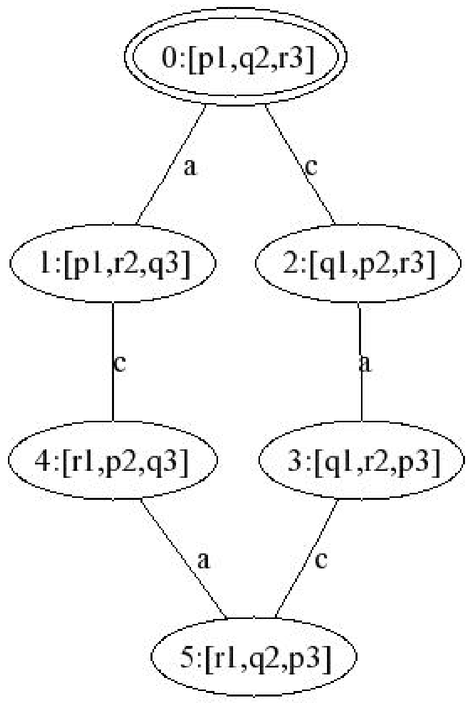


The result of updating with this:

```
result   = upd cards0 revealAB
```

```
DEMO> showM result
==> [0]
[0,1,2,3,4,5]
(0,[p1,q2,r3])(1,[p1,r2,q3])(2,[q1,p2,r3])(3,[q1,r2,p3])(4,[r1,p2,q3])
(5,[r1,q2,p3])
(a,[[0,1],[2,3],[4,5]])
(b,[[0],[1],[2],[3],[4],[5]])
(c,[[0,2],[1,4],[3,5]])
```

Viewed as a graph:



We have enough machinery now to handle quite subtle actions. Suppose the three cards are dealt to a, b, c but are still undisclosed on the table. We now want to find an action model for the action of a inspecting her own card, with the others looking on. Here it is:

```
DEMO> showM (reveal [a] [p1,q1,r1])
==> [0,1,2]
[0,1,2]
(0,p1)(1,q1)(2,r1)
(a,[[0],[1],[2]])
(b,[[0,1,2]])
(c,[[0,1,2]])
```

And here is the action of a picking up her card and showing it to the others, without taking a look herself:

```
DEMO> showM (reveal [b,c] [p1,q1,r1])
==> [0,1,2]
[0,1,2]
(0,p1)(1,q1)(2,r1)
(a,[[0,1,2]])
(b,[[0],[1],[2]])
(c,[[0],[1],[2]])
```

25 Finding Axiom Schemes for Logics of Communication

Find an axiom scheme for the interaction of updates with epistemic preconditions and epistemic formulas:

```
DEMO> tr (Up (public (K a p)) (K a p))
[C[?[a]p,a]]v[p,&[-p,-[a]p]]
DEMO> tr (Up (public (K a p)) (Up (public (K a p)) (K a p)))
[U[C[?[a]p,C[?[a]p,a]]]]v[p,&[-p,[a]p,-[a]p],&[-p,-[a]p]]
DEMO> step (public (K a p)) (Ag a) p
[C[?[a]p,a]]A[0]p
DEMO> step (public (K a p)) (Ag a) q
[C[?[a]p,a]]A[0]q
```

Find an axiom scheme for the interaction of public announcement and common knowledge:

```
DEMO> tr (Up (public p) (Pr (Star (Ags [a,b])) p))
T
DEMO> tr (Up (public p) (Pr (Star (Ags [a,b])) q))
[(U[?T,C[?p,[a,b]]])*]v[&[p,q],-p]
DEMO> step (public p) (Star (Ags [a,b])) q
[(U[?T,C[?p,[a,b]]])*]A[0]q
```

Find an axiom scheme for the interaction of secret group communication (email CC) and common knowledge:

```
DEMO> tr (Up (secret [a,b] p) (Pr (Star (Ags [a,b])) p))
[C[C[(U[?T,C[?p,[a,b]]])*,?-T],(U[U[?T,[a,b]],C[?-T,(U[?T,C[?p,[a,b]]])*,?-T]])*]]p
DEMO> step (secret [a,b] p) (Star (Ags [a,b])) p
&[[C[C[(U[?T,C[?p,[a,b]]])*,?-T],(U[U[?T,[a,b]],C[?-T,(U[?T,C[?p,[a,b]]])*,?-T]])*]]A[1]p,
[U[(U[?T,C[?p,[a,b]]])*,C[C[(U[?T,C[?p,[a,b]]])*,?-T],
(U[U[?T,[a,b]],C[?-T,(U[?T,C[?p,[a,b]]])*,?-T]])*,
C[?-T,(U[?T,C[?p,[a,b]]])*]]]]A[0]p]
DEMO> tr (Up (secret [a,b] p) (Pr (Star (Ags [a,b])) q))
&[[C[C[(U[?T,C[?p,[a,b]]])*,?-T],(U[U[?T,[a,b]],C[?-T,(U[?T,C[?p,[a,b]]])*,?-T]])*]]q,
[U[(U[?T,C[?p,[a,b]]])*,C[C[(U[?T,C[?p,[a,b]]])*,?-T],
(U[U[?T,[a,b]],C[?-T,(U[?T,C[?p,[a,b]]])*,?-T]])*,
C[?-T,(U[?T,C[?p,[a,b]]])*]]]]v[&[p,q],-p]]
```

Find an axiom scheme for the interaction of group announcement and common knowledge:

```
DEMO> tr (Up (groupM [a,b] p) (Pr (Star (Ags [b,c])) q))
&[[C[C[(U[?T,C[?p,[b,c]]])*,C[?p,[c]]],(U[U[?T,[b,c]],C[c,(U[?T,C[?p,[b,c]]])*,
C[?p,[c]]]])*]]q,[U[(U[?T,C[?p,[b,c]]])*,C[C[(U[?T,C[?p,[b,c]]])*,C[?p,[c]]],
(U[U[?T,[b,c]],C[c,(U[?T,C[?p,[b,c]]])*,C[?p,[c]]]])*,C[c,(U[?T,C[?p,[b,c]]])*]]]]v[&[p,q],-p]]
```

And so on.

Acknowledgement This paper and the tool that it describes were prompted by a series of questions voiced by Johan van Benthem in his talk at the annual meeting of the Dutch Association for Theoretical Computer Science, in Utrecht, on March 5, 2004. Thanks to Johan van Benthem, Hans van Ditmarsch, Barteld Kooi and Ji Ruan for valuable feedback and inspiring discussion.

A Special Treatment for S5, KD45 and K45

In the introduction we saw in what sense S5, KD45 and K45 are special in an epistemic update setting. This appendix proves some useful facts about relations that we will employ to give S5, KD45 and K45 models special treatment.

Since equivalence relations are euclidean and serial, every S5 model is a KD45 model. Also, obviously, every KD45 model is a K45 model. This means that we can combine the tests for S5, KD45, and K45. For that, we first explore the relation between S5, KD45 and K45 in some detail.

If R is a relation, then a point x is called *isolated in* R if $\neg\exists y\ yRx$ and $\neg\exists y\ xRy$.

Theorem 7 *Removal of the isolated points from a K45 relation creates a KD45 relation.*

Proof. Suppose R is euclidean and transitive on X. We have to show that R is euclidean, transitive and serial on $Y = X - \{x|\neg\exists y\ yRx \wedge \neg\exists y\ xRy\}$. Euclideanness and transitivity are obvious. For seriality, take $y \in Y$. By definition of Y, y is not isolated for R. So either there is a $z \in Y$ with yRz or there is a $z \in Y$ with zRy. In the first case we are done. In the second case, yRy follows from zRy by euclideanness. □

Theorem 8 *Adding a set of isolated points to a KD45 relation creates a K45 relation.*

Proof. Isolated points are trivially transitive and euclidean. □

If R is a relation, then a pair $(x, y) \in R$ is called an *entry pair* if $(y, x) \notin R$.

Theorem 9 *Removing the set of entry pairs from a KD45 relation creates an equivalence relation.*

Proof. Let R be transitive, euclidean and serial on X. We have to show that

$$S = R - \{(x, y) \mid xRy \wedge \neg yRx\}$$

is an equivalence on X.

- S is transitive: Assume xSy, ySz. Since $S \subseteq R$, it follows that xRy, yRz, and by transitivity of of R, xRz. By definition of S, it follows from xSy that yRx and from ySz that zRy. Again by transitivity of R, zRx. From xRz, zRx and the definition of S: xSz.
- S is symmetric: immediate from the definition.
- S is reflexive: Assume $x \in X$. By seriality of R, there is a $y \in X$ with xRy. By symmetry and transitivity of R, xRx. By definition of S, xSx.

□

If $\sim$ is an equivalence on X, $x \in X$ and $y \notin X$, then let

$$y_{\sim x} = \{(y, z) \mid z \in [x]_\sim\}.$$

The pairs in $y_{\sim x}$ are the entry-pairs from y to the members of the $[x]_\sim$ block of the partition induced by $\sim$. Adding such sets of entry-pairs to an equivalence creates a KD45 relation. More precisely:

Theorem 10 *If* $\sim$ *is an equivalence on* X, $X \cap Y = \emptyset$, *and* $f : Y \to X$, *then*

$$\sim \cup \bigcup\{y_{\sim f(y)} \mid y \in Y\}$$

is a KD45 relation on $X \cup Y$.

Proof. Suppose $\sim$ is an equivalence on X, $Y \cap X = \emptyset$, and $f : Y \to X$. Let

$$R = \sim \cup \bigcup \{y_{\sim f(y)} \mid y \in Y\}.$$

We have to show that R is euclidean, serial and transitive.

- R is euclidean: Let xRy and xRz. We have to show yRz. Suppose $x \in X$. Then, by construction of R, $x \sim y$ and $x \sim z$. Thus $y \sim z$ by euclideanness of $\sim$, and hence yRz. Suppose $x \notin X$. Then $xRy \in x_{\sim f(x)}$ and $xRz \in x_{\sim f(x)}$, and therefore $y, z \in [f(x)]_\sim$. It follows that $y \sim z$, and hence yRz.
- R is serial: immediate from the construction of R.
- R is transitive. Similar to the reasoning for euclideanness.

□

The upshot of the above is the following:

- Any S5 relation can be represented by the partition it induces.
- Any KD45 relation can be represented by a barbed partition (a partition where blocks may be barbed by loose entry points) or a set of balloons (a partition where each partition block is held on a string of the entry points into the block).
- Any K45 relation can be represented by a barbed partition plus a set of isolated points, or by a set of balloons plus a set of isolated points.

Representing sets as lists, we can use the type `[[a]]` (the type of lists of lists) for partitions, we can use the type `[([a],[a])]` (the type of lists of list pairs) for sets of balloons, and the type `([a],[([a],[a])])` (the type of pairs consisting of a list and a list of list pairs) for sets of isolated points together with sets of balloons.

B The DPLL prover

Implementation of Davis, Putnam, Logemann, Loveland (DPLL) theorem proving [8, 9] for propositional logic. The implementation uses discrimination trees or *tries*, following [30].

B.1 Module Declaration

```
module DPLL

where

import List
```

B.2 Clauses, Clause Sets

If we let variables be represented by their indices, with the convention that positive indices represent positive literals and the negation of an index represents the negation of the variable, then clauses can be represented as integer lists, and clause sets as lists of integer lists.

```
type Clause    = [Integer]
type ClauseSet = [Clause]
```

A valuation is simply a list of integers:

```
type Valuation = [Integer]
```

Reorder the literals in a clause to make sure that lowest variable indices are listed first. Also, throw out duplicate literals from clauses and duplicate clauses from clause sets.

```
rSort :: ClauseSet -> ClauseSet
rSort = (srt1.nub) . (map (srt2. nub))
  where srt1 = sortBy cmp
        cmp [] (_:_)  = LT
        cmp [] []     = EQ
        cmp (_:_) []  = GT
        cmp (x:xs) (y:ys) | (abs x) < (abs y)  = LT
                          | (abs x) > (abs y)  = GT
                          | (abs x) == (abs y) = cmp xs ys
        srt2 = sortBy (\ x y -> compare (abs x) (abs y))
```

A clause is trivial if the clause contains both positive and negative occurrences of some literal.

```
trivialC :: Clause -> Bool
trivialC [] = False
trivialC (i:is) = elem (-i) is || trivialC is
```

The function `clsNub` removes trivial clauses from a clause set.

```
clsNub :: ClauseSet -> ClauseSet
clsNub = filter (not.trivialC)
```

B.3 Tries

The datatype of discrimination trees.

```
data Trie = Nil | End | Tr Integer Trie Trie Trie deriving (Eq,Show)
```

`Nil` is the empty clause set, `End` a marker for a clause end.

In a trie of the form (Tr, v, P, N, R), P encodes a clause set $\{P_1, \dots, P_n\}$ representing $\{v \vee P_1, \dots, v \vee P_n\}$, N a clause set $\{N_1, \dots, N_m\}$ representing $\{\neg v \vee N_1, \dots, \neg v \vee N_m\}$, and R a clause set $\{R_1, \dots, R_k\}$ with none of the R_i containing occurrences of v.

If the clause set corresponding to P equals $\square$, and the clause set corresponding to N equals $\square$, this means that both $\{v\}$ and $\{\neg v\}$ occur in the clause set, i.e., that the clause set is a contradiction (equals $\square$).

If the clause sets corresponding to P and Q are both empty, this means that v does not occur (positively or negatively) in the clause set.

If the clause set corresponding to R equals $\square$, this means that the whole clause set equals $\square$. If the clause set corresponding to R is empty, this means that the clause set does not contain clauses without positive or negative occurrences of v.

Bearing this in mind, the following operation can be used to perform some simplifications.

```
nubT :: Trie -> Trie
nubT (Tr v p n End)   = End
nubT (Tr v End End r) = End
nubT (Tr v Nil Nil r) = r
nubT tr               = tr
```

The trie merge operation (conjunction of the corresponding clause sets):

```
trieMerge :: Trie -> Trie -> Trie
trieMerge End _ = End
trieMerge _ End = End
trieMerge t1 Nil = t1
trieMerge Nil t2 = t2
trieMerge t1@(Tr v1 p1 n1 r1) t2@(Tr v2 p2 n2 r2)
  | v1 == v2 = (Tr
                v1
                (trieMerge p1 p2)
                (trieMerge n1 n2)
                (trieMerge r1 r2)
                )
  | v1 < v2 = (Tr
                v1
                p1
                n1
                (trieMerge r1 t2)
                )
  | v1 > v2 = (Tr
                v2
                p2
                n2
                (trieMerge r2 t1)
                )
```

Assuming clauses are r-sorted, mapping clause sets into ordered tries is done as follows:

```
cls2trie :: ClauseSet -> Trie
cls2trie []            = Nil
cls2trie ([]:_)        = End
cls2trie cls@((i:is):_) =
  let j = abs i in
   (Tr
    j
    (cls2trie [ filter (/= j)  cl | cl <- cls, elem   j  cl ])
    (cls2trie [ filter (/= -j) cl | cl <- cls, elem (-j) cl ])
    (cls2trie [ cl | cl <- cls, notElem j cl, notElem (-j) cl ])
   )
```

Tries are mapped back into clause sets by:

```
trie2cls :: Trie -> ClauseSet
trie2cls Nil = []
trie2cls End = [[]]
trie2cls (Tr i p n r) =
    [ i:rest | rest <- trie2cls p ]
    ++
    [ (-i):rest | rest <- trie2cls n ]
    ++
    trie2cls r
```

B.4 Unit Subsumption and Unit Resolution

Unit clause detection in the trie format; the following function finds all unit clauses:

```
units :: Trie -> [Integer]
units Nil  = []
units End  = []
units (Tr i End n r) = i : units r
units (Tr i p End r) = -i: units r
units (Tr i p n r)     = units r
```

Unit propagation:

```
unitProp :: (Valuation,Trie) -> (Valuation,Trie)
unitProp (val,tr) = (nub (val ++ val’), unitPr val’ tr)
  where
  val’ = units tr
  unitPr :: Valuation -> Trie -> Trie
  unitPr [] tr = tr
  unitPr (i:is) tr = unitPr is (unitSR i tr)
```

Unit subsumption and resolution.

```
unitSR :: Integer -> Trie -> Trie
unitSR i = (unitR pol j) . (unitS pol j)
  where pol = i>0
        j   = abs i
```

Unit subsumption: delete every clause containing the literal. The literal is encoded as `(pol,i)`, where `pol` gives the sign and `i` is a positive integer giving the index of the variable.

```
unitS :: Bool -> Integer -> Trie -> Trie
unitS pol i Nil = Nil
unitS pol i End = End
unitS pol i tr@(Tr j p n r) | i == j = if pol
                                       then nubT (Tr j Nil n r)
                                       else nubT (Tr j p Nil r)
                            | i < j  = tr
                            | i > j  = nubT (Tr
                                        j
                                        (unitS pol i p)
                                        (unitS pol i n)
                                        (unitS pol i r)
                                       )
```

Unit resolution: delete the mate of the literal from every clause containing it.

```
unitR ::  Bool -> Integer -> Trie -> Trie
unitR pol i Nil = Nil
unitR pol i End = End
unitR pol i tr@(Tr j p n r) | i == j = if pol
                                       then
                                       nubT (Tr
                                        j
                                        p
                                        Nil
                                        (trieMerge n r)
                                       )
                                       else
                                       nubT (Tr
                                        j
                                        Nil
                                        n
                                        (trieMerge p r)
                                       )
                            | i < j  = tr
                            | i > j  = nubT (Tr
                                        j
                                        (unitR pol i p)
                                        (unitR pol i n)
                                        (unitR pol i r)
                                       )
```

B.5 Splitting

To treat splitting, we need functions for setting variables to true and false, respectively. Setting a variable to true is done by:

```
setTrue :: Integer -> Trie -> Trie
setTrue i Nil = Nil
setTrue i End = End
setTrue i tr@(Tr j p n r) | i == j = trieMerge n r
                          | i < j  = tr
                          | i > j  = (Tr
                                      j
                                      (setTrue i p)
                                      (setTrue i n)
                                      (setTrue i r)
                                     )
```

Setting a variable to false is done similarly:

```
setFalse :: Integer -> Trie -> Trie
setFalse i Nil = Nil
setFalse i End = End
setFalse i tr@(Tr j p n r) | i == j = trieMerge p r
                           | i < j  = tr
                           | i > j  = (Tr
                                      j
                                      (setFalse i p)
                                      (setFalse i n)
                                      (setFalse i r)
                                     )
```

Always split on the first variable:

```
split :: (Valuation,Trie) -> [(Valuation,Trie)]
split (v,Nil) = [(v,Nil)]
split (v,End) = []
split (v, tr@(Tr i p n r)) = [(v++[i], setTrue i tr),
                              (v++[-i],setFalse i tr)]
```

B.6 DPLL

Davis, Putnam, Loveland, Longeman (DPLL): keep splitting after unit propagation.

```
dpll :: (Valuation,Trie) -> [(Valuation,Trie)]
dpll (val,Nil) = [(val,Nil)]
dpll (val,End) = []
dpll (val,tr) =
  concat [ dpll vt | vt <- (split.unitProp) (val,tr) ]
```

Wrap it all up:

```
dp :: ClauseSet -> [(Valuation,Trie)]
dp cls = dpll ([], (cls2trie . rSort) (clsNub cls))
```

References

[1] Baltag, A. A logic for suspicious players: epistemic action and belief-updates in games. *Bulletin of Economic Research 54*, 1 (2002), 1–45.

[2] Baltag, A., Moss, L., and Solecki, S. The logic of public announcements, common knowledge, and private suspicions. Tech. Rep. SEN-R9922, CWI, Amsterdam, 1999.

[3] Baltag, A., Moss, L., and Solecki, S. The logic of public announcements, common knowledge, and private suspicions. Tech. rep., Dept of Cognitive Science, Indiana University and Dept of Computing, Oxford University, 2003.

[4] Benthem, J. v. Language, logic, and communication. In *Logic in Action*, J. van Benthem, P. Dekker, J. van Eijck, M. de Rijke, and Y. Venema, Eds. ILLC, 2001, pp. 7–25.

[5] Benthem, J. v. One is a lonely number: on the logic of communication. Tech. Rep. PP-2002-27, ILLC, Amsterdam, 2002.

[6] Blackburn, P., de Rijke, M., and Venema, Y. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[7] Chellas, B. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[8] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[9] Davis, M., and Putnam, H. A computing procedure for quantification theory. *Journal of the ACM 7*, 3 (1960), 201–215.

[10] Ditmarsch, H. v. *Knowledge Games*. PhD thesis, ILLC, Amsterdam, 2000.

[11] Eijck, J. v. Communicative actions. CWI, Amsterdam, 2004.

[12] Eijck, J. v. Reducing dynamic epistemic logic to PDL by program transformation. CWI, Amsterdam, `www.cwi.nl:~/papers/04/delpdl/`, 2004.

[13] Eijck, J. v., and Ruan, J. Action emulation. CWI, Amsterdam, `www.cwi.nl:~/papers/04/ae`, 2004.

[14] Fagin, R., Halpern, J., Moses, Y., and Vardi, M. *Reasoning about Knowledge.* MIT Press, 1995.

[15] Gerbrandy, J. *Bisimulations on planet Kripke.* PhD thesis, ILLC, 1999.

[16] Gerbrandy, J. Dynamic epistemic logic. In *Logic, Language and Information, Vol. 2*, L. e. a. Moss, Ed. CSLI Publications, Stanford, 1999.

[17] Goldblatt, R. *Logics of Time and Computation, Second Edition, Revised and Expanded*, vol. 7 of *CSLI Lecture Notes.* CSLI, Stanford, 1992 (first edition 1987). Distributed by University of Chicago Press.

[18] Harel, D., Kozen, D., and Tiuryn, J. *Dynamic Logic. Foundations of Computing.* MIT Press, Cambridge, Massachusetts, 2000.

[19] Hintikka, J. *Knowledge and Belief: An Introduction to the Logic of the Two Notions.* Cornell University Press, Ithaca N.Y., 1962.

[20] Hollenberg, M. *Logic and Bisimulation.* PhD thesis, Utrecht University, 1998.

[21] J.E.Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Academic Press, 1971.

[22] Jones, S. P., Hughes, J., et al. Report on the programming language Haskell 98. Available from the Haskell homepage: `http://www.haskell.org`, 1999.

[23] Knuth, D. *Literate Programming.* CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[24] Kooi, B., and van Benthem, J. Reduction axioms for epistemic actions. In *AiML-2004: Advances in Modal Logic* (2004), R. Schmidt, I. Pratt-Hartmann, M. Reynolds, and H. Wansing, Eds., no. UMCS-04-9-1 in Technical Report Series, University of Manchester, pp. 197–211.

[25] Kooi, B. P. *Knowledge, Chance, and Change.* PhD thesis, Groningen University, 2003.

[26] Koutsofios, E., and North, S. Drawing graphs with *dot.* Available from `http://www.research.att.com/~north/graphviz/`.

[27] Paige, R., and Tarjan, R. E. Three partition refinement algorithms. *SIAM J. Comput. 16*, 6 (1987), 973–989.

[28] Papadimitriou, C. *Computational Complexity.* Addison-Wesley, 1994.

[29] Ruan, J. Exploring the update universe. Master's thesis, ILLC, Amsterdam, 2004.

[30] Zhang, H., and Stickel, M. E. Implementing the Davis-Putnam method. *Journal of Automated Reasoning 24*, 1/2 (2000), 277–296.