Correspondence theory for implication in LP
(Bachelorproject)

Thomas Derksen, s2016346, t.w.derksen@student.rug.nl,
Barteld Kooi*

25 augustus 2014

Abstract

In traditional logic, there seem to be some problems with implication. Especially in many-valued logics, including Graham Priest's Logic of Paradox (LP), the validity of some inferences concerning implication can be different from human intuition about them. If you want to make these inferences invalid in LP, it can have an impact on the truth-table for implication. In this paper, I will do some research into implication in LP. First, I take a look at the correspondence between ten problematic inferences concerning implication and the truth-table for implication in LP. I will show that it is possible to make nine of these invalid by defining the truth-table for implication in certain ways. I will then describe a program that can convert any given inference in LP into a set of restrictions for the truth-table of implication. This program can be used to form the basis for a correspondence theory for implication in LP.

1 Introduction

In 1979, Graham Priest wrote a paper in which he introduced a three-valued logic called the Logic of Paradox (LP) (Priest (1979)). This logic has the truth-values 1 (true), 0 (false) and i (both true and false). In LP, both 1 and i are designated values, which means that these are the values that are preserved in valid inferences. The idea behind LP was that it didn't try to avoid the various logical paradoxes that were (and are) around. Rather, Priest acknowledged that there are things that can be true and not true at the same time. But, as Priest also noted in his original paper, LP has some problems with the implication operator ($\rightarrow$). If you define $A \rightarrow B$ as $\neg A \vee B$, you have to give up modus ponens, modus tollens and reductio ad absurdum. However, defining it in any other way isn't really straightforward either. Because of this, implication is somewhat problematic in LP.
In this paper, I am going to take a look at different forms of implication for LP. First, I will look at a few well known problematic inferences with implication (which are explained in some detail in Priest (2001)). These are:

1) $q \models p \rightarrow q$
2) $\neg p \models p \rightarrow q$
3) $(p \wedge q) \rightarrow r \models (p \rightarrow r) \vee (q \rightarrow r)$
4) $(p \rightarrow q) \wedge (r \rightarrow s) \models (p \rightarrow s) \vee (r \rightarrow q)$
5) $\neg(p \rightarrow q) \models p$
6) $p \rightarrow r \models (p \wedge q) \rightarrow r$
7) $p \rightarrow q, q \rightarrow r \models p \rightarrow r$
8) $p \rightarrow q \models \neg q \rightarrow \neg p$
9) $\models p \rightarrow (q \vee \neg q)$
10) $\models (p \wedge \neg p) \rightarrow q$

Although these inferences are completely valid in most many-valued logics, they aren't very coherent with the intuition of most people. Take inference 5, for example. If you use this in a real life application, you could get something like the following: *It is not true that, if it is raining, dogs fall out of the sky. Therefore, it is raining.* As you can see, this argument doesn't make a lot of sense, even though it is logically valid. Something similar goes for the other inferences. In all of these cases, the logicial validity of the inferences isn't the same as the intuition most people have about them.
If you want any of these to be invalid in LP, it has some consequences for the truth-table of implication, and some cells will have to take

*University of Groningen, Department of Philosophy

certain values. Thus, it is possible to find correspondences between these different inferences and implication in LP. Not all of the resulting forms of implication are equally useful, and some don't make much sense at all. Therefore, I will also look at some of the forms of implication and their usefulness. In an article from 2012, Allard Tamminga and Barteld Kooi showed that each entry in a truth table for any binary operator is characterized by an inference scheme, and that this also works the other way round. (Kooi and Tamminga (2012)). Using the inferences above, I will try to generalize this, by showing that the ten inferences are characterized by certain restrictions on the truth-table for the binary operator of implication. To generalize it even further, I will describe a computer program that will convert any given inference from LP into a set of restrictions on the truth table for implication. This can be used to show direct correspondence between these inferences and implication in LP. If you would want a more intuitive form of LP, it would be possible to use this program to define implication such that certain counter-intuitive inferences are invalidated.

The results from the ten inferences and the program will form a basis for a correspondence theory of LP. This kind of correspondence theory has previously mainly been researched for modal logic. In 1975, Henrik Sahlqvist published a paper about correspondence for the first and second order semantics for modal logic. In this paper, he introduced the so called Sahlqvist formulas, which are modal formulas that correspond to first order classes of Kripke frames (Sahlqvist). A little bit later, in 1976, Johan van Benthem used correspondence theory to show that some complex modal axioms correspond to simple classical requirements (van Benthem (1976)). He also showed the relation to correspondence theory and algebra. Research for correspondence theory in LP however, is hard to find.

In this paper, I will try to do something about this. I will first devote a chapter to the ten problematic inferences above. Then, I will describe the program that can convert inferences to restrictions on the truth-table for implication. I will conclude by summarizing what I found, and what could be improved or researched further.

2 Ten problematic inferences concerning implication

Let us now look at the ten problematic inferences concerning implication that were introduced before. Although they are valid in some (if not all) many-valued logics, they all pose some problems because they are counter-intuitive in some way. They make clear that implication can be quite problematic in these logics. Therefore, it would make sense to see if, and how, we can invalidate these inferences in LP. As we will see later on, not all of these inferences can be invalidated, no matter what form of implication we use. In particular, $(p \wedge q) \rightarrow r \not\models (p \rightarrow r) \vee (q \rightarrow r)$ will unavoidably be valid. If you want any of the other inferences to be invalid, some of the entries in the truth-table for implication will have to take certain values, which will make certain patterns emerge.

An inference scheme is said to be valid if and only if for each valuation it holds that if the antecedent of the inference is not 0, then the predescent is not 0. In LP, these inference schemes consist of atomic formulas using negation ($\neg$), disjunction ($\vee$) and conjunction ($\wedge$). A valuation on these inference schemes can then be formed according to the truth-tables for $\neg$, $\vee$ and $\wedge$:

$f_\neg$	
0	1
i	i
1	0

$f_\vee$	0	i	1
0	0	i	1
i	i	i	1
1	1	1	1

$f_\wedge$	0	i	1
0	0	0	0
i	0	i	i
1	0	i	1

So, if an inference $\varphi \not\models \psi$ is invalid, it has to be the case that $\varphi \in \{i, 1\}$ and $\psi = 0$. Now suppose that $\varphi = a \rightarrow b$ and $\psi = b$. The resulting inference will be $a \rightarrow b \not\models b$. If we want this inference to be invalid in LP, it follows that $v(a \rightarrow b) \in \{i, 1\}$ and $v(b) = 0$. Since $v(b) = 0$, it also means that $v(a \rightarrow 0) \in \{i, 1\}$. Thus, in the truth table for implication, there has to be an i or a 1 in at least one of the cells in the 0-column. So, for example, if $f_\rightarrow(1, 0) = 0$, this inference would be invalid. It is possible to do something similar for the invalidation of the ten problematic inferences:

1) $q \not\models p \rightarrow q$:
Example: *If the sun is shining. it doesn't have to be the case that when it's monday, the sun is shining*
Restrictions: *there has to be at least one 0 in the 1-column or in the i-column*
Proof: suppose $v(q)$ is designated and $v(p \rightarrow q) = 0$. Then, $f_{\rightarrow}(v(p), 1)$ or $f_{\rightarrow}(v(p), i)$ has to be 0. Now suppose that there is a zero in the 1-column or i-column, so $v(\varphi \rightarrow i)$ or $v(\varphi \rightarrow 1)$ is 0. If $v(p) = \varphi$ and $v(q)$ is 1 or i, $v(q)$ is designated while $v(p \rightarrow q) = 0$ and the inference is invalid.

2) $\neg p \not\models p \rightarrow q$:
Example: *If it is not tuesday, it doesn't have to be the case that when it's tuesday the sun is shining*
Restrictions: *there has to be at least one 0 in the 0-row or the i-row*
Proof: Suppose $v(\neg p) = 1$ or $v(\neg p) = i$. Then, $v(p) = 0$ or $p = i$. Also suppose, $v(p \rightarrow q) = 0$. Then, $f_{\rightarrow}(0, v(q)) = 0$ or $f_{\rightarrow}(i, v(q)) = 0$. Now suppose there is a 0 in the 0-row or the i-row. Then, if $v(p)$ is 0 or i, $v(\neg p)$ is 1 or i, while $v(p \rightarrow q) = 0$, so the inference is invalid.

3) $(p \wedge q) \rightarrow r \not\models (p \rightarrow r) \vee (q \rightarrow r)$:
Example: *If it is the case that something with a beard and a red pointy hat is a gnome, it doesn't mean that something with a beard is a gnome or something with a pointy hat is a gnome*
Restrictions: *if you want to invalidate this inference, the resulting logic is not truth-functional*
Proof: Suppose that $v((p \wedge q) \rightarrow r)$ is designated and $v((p \rightarrow r) \vee (q \rightarrow r))$ is 0. This means that $v(p \rightarrow r) = 0$ and $v(q \rightarrow r) = 0$. Now suppose that v(p) and v(q) both have the same value φ. Then, whatever that value is, $v(p \wedge q)$ will also be φ. Therefore, $v((p \wedge q) \rightarrow r)$ will be 0. But if we suppose that v(p) and v(q) have different values φ and ψ, where $v(\varphi \rightarrow r) = 0$ and $v(\psi \rightarrow r) = 0$, $v(p \wedge q)$ will always take the value of either v(p) or v(q). This also means that $v((p \wedge q) \rightarrow r)$ will be 0.

4) $(p \rightarrow q) \wedge (r \rightarrow s) \not\models (p \rightarrow s) \vee (r \rightarrow q)$:
Example: *The facts that raining makes you wet and falling hurts don't imply that raining hurts and falling makes you wet*
Restrictions: *there have to be 4 cells in the truth-table that form a rectangle, with designated values on two opposing corners, and zeros on the other two corners.*
Proof: Suppose $v((p \rightarrow q) \wedge (r \rightarrow s))$ is designated, so $v(p \rightarrow q)$ is designated and $v(r \rightarrow s)$ is designated. Also suppose, $v((p \rightarrow s) \vee (r \rightarrow q)) = 0$, so $v(p \rightarrow s) = 0$ and $v(r \rightarrow q) = 0$. It follows that $v(s) \neq v(q)$ and $v(p) \neq v(r)$. This means that there have to be four values, φ, ψ, τ and δ, where $f_{\rightarrow}(v(\varphi), v(\psi))$ is 0, $f_{\rightarrow}(v(\varphi), v(\tau))$ is designated, $f_{\rightarrow}(v(\delta), (\psi))$ is designated and $f_{\rightarrow}(v(\delta), v(\tau))$ is zero. Now suppose the truth table has 4 cells in a rectangle where the corners are $f_{\rightarrow}(v(\varphi), v(\pi))$ and $f_{\rightarrow}(v(\varphi), v(\tau))$, which are designated, and $f_{\rightarrow}(v(\psi), v(\pi))$ and $f_{\rightarrow}(v(\psi), v(\tau))$, which are 0. Then, if $v(p) = v(\varphi)$, $v(q) = v(\pi)$, $v(r) = v(\psi)$ and $v(s) = v(\tau)$, $v(p \rightarrow q)$ and $v(r \rightarrow s)$ are both designated, so the left side of the inference is designated, and $v(p \rightarrow r)$ and $v(q \rightarrow r)$ are both 0, so the right side of the inference is 0. This means that the inference is invalid.

5) $\neg(p \rightarrow q) \not\models p$:
Example: *It is not true that, when it rains, dogs fall out of the sky. This doesn 't mean that it is raining*
Restrictions: *there has to be at least one 0 or i in the 0-row*
Proof: Suppose $v(\neg(p \rightarrow q))$ is 1 or i, so $v(p \rightarrow q)$ is 0 or i. Also, $v(p) = 0$, so $f_{\rightarrow}(0, v(q))$ is 0 or i. Now suppose that there is a 0 or an i in the 0-row. Then, if $v(p) = 0$, $v(p \rightarrow q)$ is i or 0, so $v(\neg(p \rightarrow q)$ is i or 1. This means that the inference is invalid.

6) $p \rightarrow r \not\models (p \wedge q) \rightarrow r$:
Example: *If the weather is nice you go to the park. This doesn't mean that if the weather is nice and your leg falls of you go to the park*
Restrictions: *there has to be a column where the 1-row is designated while the 0-row or the i-row are 0, or a column where the i-row is designated while the 0-row is 0*
Proof: Suppose $v(p \rightarrow r)$ is designated and $v((p \wedge q) \rightarrow r)$ is 0. It can then be concluded that $v(p) \neq v(p \wedge q)$. If $v(p) = 0$, it also follows that $v(p \wedge q) = 0$. So, v(p) is either 1 or i. If $v(p) = 1$, $v(p \wedge q)$ can be 0 or i. If $v(p) = i$, $v(p \wedge q)$ can be 0. Now suppose that there is a column where the 1-row is designated while the 0-row or the i-row is 0. Suppose this column is $f_{\rightarrow}(\varphi, v(r))$. Now, if $v(p) = 1$, $v(p \wedge q)$ can be 0 or i, so $v((p \wedge q) \rightarrow r)$ can be 0. If $v(p) = i$, $v(p \wedge q)$ can be 0, so $v((p \wedge q) \rightarrow r)$

can be 0. This would make the inference invalid.

7) $p \to q$, $q \to r \not\models p \to r$:
Example: *If the other candidates pull out, you get the job. If you get the job, the other candidates are dissapointed. This doesn't mean that if the other candidates pull out, they are dissapointed*
Restrictions: *there have to be three values φ, ψ and τ where $\varphi \to \psi$ is designated, $\psi \to \tau$ is designated and $\varphi \to \tau$ is 0. $v(\varphi)$ and $v(\psi)$ have to be different and $v(\psi)$ and $v(\tau)$ have to be different*
Proof: Suppose $v(p \to q)$ is designated, $v(q \to r)$ is designated and $v(p \to r)$ is 0. $v(p \to r) \neq v(q \to r)$, so it follows that $v(p) \neq v(q)$. Also, $v(p \to q) \neq v(p \to r)$, so $v(q) \neq v(r)$. Now suppose there are three values φ, ψ and τ where $\varphi \to \psi$ is designated, $\psi \to \tau$ is designated and $\varphi \to \tau$ is 0. If $v(p) = v(varphi)$, $v(q) = v(psi)$ and $v(r) = v(tau)$. Then, $v(p \to q)$ and $v(q \to r)$ are designated while $v(p \to r)$ is 0, so the inference is invalid.

8) $p \to q \not\models \neg q \to \neg p$:
Example: *We suppose that if something is black, it is a crow. This doesn't mean that when something is not black, it is not a crow*
Restrictions: *there have to be at least two cells on a diagonal line in the truth-table where one is designated and the other is zero. The i-row in the i-column can not be one of those cells*
Proof: Suppose $p \to q$ is designated and $\neg q \to \neg p$ is 0. It follows that it can't be that $\neg p = q$ and $\neg q = p$, so it can't be that one is 1 while the other is 0. It can also not be the case that they are both i. Now suppose that there are two cells on a diagonal line in the truth-table which are both not $f_{\to}(i, i)$, where one is designated and the other is 0. Suppose $f_{\to}(v(\varphi), v(\psi))$ is designated, $v(p) = v(\varphi)$ and $v(q) = v(psi)$. If either $v(p)$ or $v(q)$ is not i, negation makes the valuation of at least one of them different. Then $v(\neg q \to \neg p)$ can be zero and the inference is invalid.

9) $\not\models p \to (q \vee \neg q)$:
Since $v(q \vee \neg q)$ is always designated, every p usually implies it.
Example: *If you like curling, this does not imply that it's raining or it's not raining*
Restrictions: *there has to be at least one 0 in the 1-column or the i-column*
Proof: Suppose $v(p \to (q \vee \neg q)) = 0$. Since $v(q \vee \neg q)$ will always be 1 or i, $f_{\to}(v(p), 1)$ or $f_{\to}(v(p), i)$ has to be 0. Now suppose there is a 0 in the 1-column or the i-column, so $f_{\to}(v(\varphi), i)$ or $f_{\to}(v(\varphi), 1)$ is 0. $v(q \vee \neg q)$ will always be 1 or i, so if $v(p) = v(\varphi)$, $v(p \to (q \vee \neg q))$ is 0 and the inference is invalid.

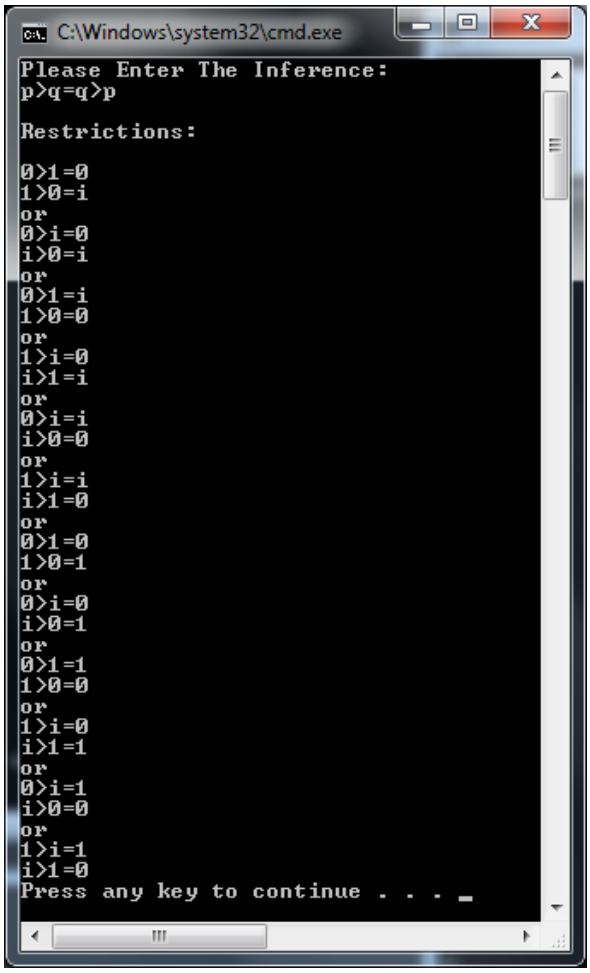
10) $\not\models (p \wedge \neg p) \to q$:
Since $v(p \wedge \neg p)$ is always 0 or i, it usually implies everything.
Example: *If water is blue and not blue, it does not imply that dolphins are mammals*
Restrictions: *there has to be at least one 0 in the 0-row or the i-row*
Proof: Suppose $v((p \wedge \neg p) \to q) = 0$. $v(p \wedge \neg p)$ will always be 0 or i, so $f_{\to}(0, v(q))$ or $f_{\to}(i, v(q))$ has to be 0. Now suppose there is a 0 in the 0-row or the i-row, so $f_{\to}(i, v(\varphi))$ or $f_{\to}(0, v(\varphi))$ is 0. $v(p \wedge \neg p)$ will always be 0 or i, so if $v(q) = v(\varphi)$, $v((p \wedge \neg p) \to q)$ is 0 and the inference is invalid.

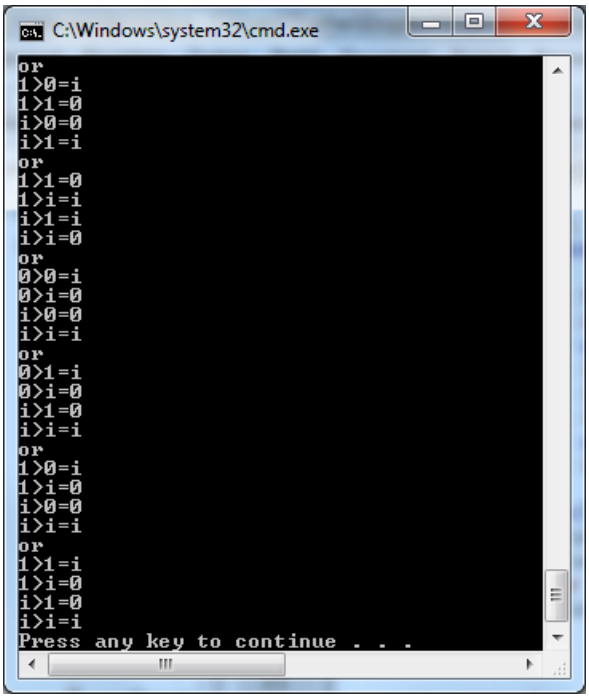
One of the things that becomes apparent is that if you want to invalidate inference 1, you'll automatically also invalidate inference 9 and vice versa. It is impossible to have a truth-table for implication which results in one being valid while the other is invalid. The same can be said about inference 2 and inference 10. Another notable result is that it is impossible to invalidate inference 3 in a truth-functional form of LP. If you'd invalidate this inference, the truth-value of arguments would not be a strictly defined function of the truth-values of operators in its sentences. Because of this, implication is bound to be problematic for LP, no matter how you define it.
Let us now take a closer look at one of the inferences, and the restrictions that follow from it. If we want to invalidate inference 5, for example, there has to be a 0 or an i in the 0-row. It might be the case that we want to invalidate inference 1 and 2 as well. Then, there should also be a 0 in the1-column or the i-column and also a 0 in the 0-row or i-row. This is easily accomplished by defining implication such that $f_{\to}(0, i)$ is 0. If it is defined like that, inference 1, 2 and 5 are not valid in LP, which might feel more intuitive in some cases. The fact that $f_{\to}(0, i)$ is 0 probably doesn 't feel that

counter-intuitive either. It might be the case that other, more intuitive inferences are invalidated by this definition of implication, which is something that could be researched further.
However, it is also possible that some restrictions give rise to truth-tables that aren't very usefull at all. Supposse that we'd want inference 1 to be invalid. It follows that there has to be at least one 0 in the 1-column or i-column. As we saw before, this constraint can be met by making $f_{\rightarrow}(0, i)$ zero. But it can also be met by saying that $f_{\rightarrow}(1, 1)$ is zero. In this case, even though the initial counter-intuitive inference is now invalid, new problems arise. Most evidently, modus ponens has to be given up. So even if a certain truth-table makes a counter-intuitive inference invalid, it doesn't necessarily mean that the resulting form of implication feels intuititive, or that it even makes sense.

3 A program for correspondence between inferences and implication

We have just seen that it is possible to invalidate inferences in LP by defining implication in a certain way. It would be interesting to see if we can find a way to do something similar for any inference with implication that we feel is counter-intuitive. This could result in a more intuitive form of implication in LP.
To do so, I wrote an algorithm in java that can do exactly that. Given that invalidating an inference doesn't result in a non-truth-functional logic, it can give restrictions that correspond to the invalidation of any given inference concerning implication in LP. The program asks the user which inference it would like to be invalid. It then lists all possible combinations of entries in the truth-table for implication that invalidate this inference.
The program works as follows:
First, it asks the user for an inference. It then parses this inference and gives the right side value 0. If there's a left side to the inference, it also creates two other sub-paths: one where the right side of the inference has value 1, and one where it has value i. It then proceeds by parsing the left and right sides of the inference and giving the left and right sides of these sub-arguments the apropriate values. For example, if the right side of the inference (with value 0) is $p \wedge q$, it will give both p and q value 0. It continues by recursively going through all of these sub-arguments this way.
If one of the sub-arguments has length 1, it checks if it is a lowercase letter. If this is the case, it stores the value of this letter in an array with all the values of all the atoms. If it isn't, the inference is not well-formed. If one of the sub-arguments is an implication, it stores this implication with the adequate value in another array. When it encounters the first implication, it notes that there is at least one. If there are no implications in the inference at all, the program doesn't print out any restrictions, but just prints out that there are no implications.
If the program encounters a sub-argument where both sides can take multiple values, it makes new paths for all of these possible values. For example, if it encounters the sub-argument $p \wedge q$ with value 0, it makes two paths, one path where the left side of the argument is zero, and another path where the right side of the argument is zero.
When the program has gone through all of the sub-arguments, it is time to look at the stored implications. For every implication of every path, the program checks what values are possible for the left and the right side of it, given the values the atoms have taken. It then gives the atoms of the implication the appropiate values. For example, if 0 and 1 are possible values for the left and right side of the implication $p \rightarrow q$, it stores 0 and 1 for the atoms p and q, respectively. It then goes on to the next implication and checks which values are possible for all of the possible values of the earlier implications. If the program has gone through the last implication of a path, it stores all possible combinations of values in an array. After it has gone through the last implication of the last path, it prints all of the possible combinations of values. Before printing a combination, it first checks if this set of answers is already printed before, in which case it doesn't print it anymore.
So, to summarize: the user gives a (possibly counter-intuitive) inference as input. The program first recursively goes through the inference and looks which values it can already find for the atoms and the implications in it. It then looks at all of the stored implications and looks at which combination of values the atoms of the implication can take. It proceeds to print the restrictions for the





truth-table for implication.
Let us look at an example:
The inference $p \to q \models q \to p$ seems quite counter-intuitive, so we want it to be invalid in LP. If we give this as input to the program, it gives us the restrictions as seen in figure 1.

One limitation of this program is that it can't adequately handle implications in implications (e.g. $(p \to q) \to q$). In its current form the program is not designed to handle inferences containing this.
Another problem this program has is the fact that can not accurately describe certain patterns. For example, if you give it $(p \to q) \wedge (r \to s) \models (p \to s) \vee (r \to q)$ as input, the output is just a list of all possible combinations of truth-table entries. The program gives all possible options of restrictions, but it can't discover the fact that there have to be 4 cells in the truth-table that form a rectangle, with designated values on two opposing corners, and zeros on the other two corners. In figure 2, you can see that the list of possible values it gives is quite a mess.

If you would want it to be able to actually find the patterns, the program would have to be way more complicated. If the inference you want to invalidate isn't overly complicated, this shouldn't actually be that much of a problem. If there is an inference concerning implication in LP that feels counter-intuitive, this program could be a quick and efficient way to see what definitions of implication could make it invalid.

4 Conclusion

In conclusion, let us look at what we have found out in the previous chapters. There are some inferences that are valid in LP even though they don't feel very intuitive. It is possible to invalidate most of these by changing the truth-table for implication. However, some counter-intuitive inferences

can not be invalidated if you want LP to be truth-functional.
The inference $(p \wedge q) \rightarrow r \not\models (p \rightarrow r) \vee (q \rightarrow r)$ will be valid in LP, regardless of the truth-table for implication. Except for that one, all of the ten problematic inferences mentioned earlier can be made invalid just by changing the truth-table for implication. This includes the paradox of entailment and other well-known paradoxes. In an example, we already saw that some of the restrictions that follow from invalidating these inferences can be easily combined in a truth-table for implication. Defining implication with such a truth-table might result in a more intuitive form of LP.
I also found that for LP, it is possible to create a program that translates the invalidation of inferences concerning implication into restrictions on the truth-table for implication. This makes it possible to invalidate inferences that feel counter-intuitive. However, some of the patterns of restrictions I described for the ten problematic inferences are hard to be described efficiently by a computer program. Still, the program can be used to quickly find out how to define implication if a certain inference should be invalid because it doesn 't feel intuitive.

5 Discussion

Now that we have a tool to convert inferences to restrictions on the truth-table for implication, the next step might be to find out which other inferences concerning implication are also counter-intuitive. It is then possible to see what invalidating them does to implication. This might ultimately lead to a more intuitive multi-valued logic that is derived from LP.
It might also be interesting to see what invalidating certain interferences does for the validity of other ones. As we saw earlier, invalidating inferences sometimes automaticaly invalidates others as well. It might also be possible that invalidating some inferences causes others to unavoidably be valid.
Even though, for this paper, I have only looked at LP, it would also be possible to look at this kind of correspondence for other (many-valued) logics. One thing that is notable is the fact that a lot of the ten problematic inferences are invalid in other three-valued logics (Priest 2001 has a nice overview of this). Almost all of them are also invalid in modal logic.
The program is designed in such a way that it is quite easy to make it work for logics that are similar to LP. For most three-valued logics, you'd mainly have to change the rules for the operators. This would make it possible to find direct correspondence between inferences and the truth-table for implication for other logics.

6 Appendix

6.1 Program code

```
import java.io.*;
import java.util.*;
class Correspondence {

	private static String[][] letters = new String[20000][26];
	private static char[] conflict = new char[20000];
	private static argument[][] infarray = new argument[20000][5];
	private static int counter = 0;
	private static int max = 1;
	private static int infaanw = 0;
	private static argument[][] imps = new argument[26][100];
	private static int cimps = 0;

	public static class argument{
		public String left;
		public String right;
		public char operator;
		public String value;

	}

	//Copies values of letters
	public static void copy(int x){
		for(int j=1;j<x;j++){;
			for(int i=0;i<26;i++){
				letters[max-j][i]=letters[counter][i];
			}
		}
	}

	//Copies values of implications
	public static void copyImps(int x){
		for(int j=1;j<x;j++){
			for(int i=0;i<cimps;i++){
				imps[max-j][i]=imps[counter][i];
			}
		}
	}

	//Gives letters a value if they don't have one yet
	public static void valuate(String str, String value){
		char ch = str.charAt(0);
		if(Character.isLetter(ch)){
			if(letters[counter][((int)ch-97)]==null){
				letters[counter][((int)ch-97)]=value;
			}else{
				if(letters[counter][((int)ch-97)]==value){
				}else{
					conflict[counter]='1';
```

```
                }
            }
        }else{ //If you try to valuate something that isn't a lowercase letter, the
            inference is not well-formed
            System.out.println("This is not a correct inference.");
            System.exit(0);
        }
}

//Parses arguments or subarguments. It is used with an operator, and checks if that
    operator is the operator of the (sub)argument
public static argument Parse(String str, String value, char operator){
        //First checks if the whole argument or subargument is enclosed in brackets, and
            if so, it ignores them
        if(str.charAt(0)=='('){
                int c=0;
                while(str.charAt(c)!=')'){
                        c++;
                }
                if(c==str.length()-1){
                        String sub = str.substring(1,(str.length()-1));
                        return Parse(sub, value, operator);
                }
        }
        argument A=new argument();
        A.value=value;
        int i=0;
        while(str.charAt(i)!=operator){
                //Skips parts in brackets
                if(str.charAt(i)=='('){
                        int n = i;
                        while(str.charAt(n)!=')'){
                                n++;

                        }
                        i=n;
                }else{
                if(i==(str.length()-1)){
                        break;
                }
                i++;}

        }
        //If it doesn't loop through the whole (sub)argument because it finds the
            operator, it stores that operator as the operator of the (sub)argument
        if(i!=(str.length()-1)){
                A.operator = operator;
                if(i>0){
                        A.left=str.substring(0,i);
                }
                A.right=str.substring((i+1),str.length());

        //If it does loop through the whole (sub)argument, it goes into the function
            with the next operator in line
```

```
        }else{
                switch(operator){
                        case '=':       return Parse(str, value, '^');

                        case '^':       return Parse(str, value, ',');

                        case ',':       return Parse(str, value, 'V');

                        case 'V':       return Parse(str, value, '>');

                        case '>':       return Parse(str, value, '!');

                        default:        if(str.length()==1){
                                                A.left = str;
                                                return A;
                                        }else{ //If the (sub)argument does not have
                                            an operator and is longer than 1
                                            character, the inference is not well-

                                                                formed
                                            System.out.println("This is not a
                                                correct inference.");
                                            System.exit(0);
                                        }
                                        break;
                }
        }
        return A;
}

public static void and(argument A){
        switch(A.value){
                //If, for example, a^b=i, it creates three subpaths, one where a is i and
                    b is i, one where a is 1 and b is i and one where a is i b is 1. It
                    does            something similar for ab=0
                case "0": infarray[max][0]=Parse(A.right,"0", '^');
                                infarray[max][1]=Parse(A.left,"1", '^');
                                conflict[counter]='s';
                                max++;
                                infarray[max][0]=Parse(A.right,"0", '^');
                                infarray[max][1]=Parse(A.left,"i", '^');
                                max++;
                                infarray[max][0]=Parse(A.right,"0", '^');
                                infarray[max][1]=Parse(A.left,"0", '^');
                                max++;
                                infarray[max][0]=Parse(A.left,"0", '^');
                                infarray[max][1]=Parse(A.right,"1", '^');
                                max++;
                                infarray[max][0]=Parse(A.left,"0", '^');
                                infarray[max][1]=Parse(A.right,"i", '^');
                                max++;
                                break;
                case "1": argument A1 = Parse(A.right, "1", '^');
                                solve(A1);
```

```
                                argument A2 = Parse(A.left, "1", '^');
                                solve(A2);
                                break;
                        case "i": infarray[max][0]=Parse(A.right,"i", '^');
                                infarray[max][1]=Parse(A.left,"1", '^');
                                conflict[counter]='s';
                                max++;
                                infarray[max][0]=Parse(A.right,"1", '^');
                                infarray[max][1]=Parse(A.left,"i", '^');
                                max++;
                                infarray[max][0]=Parse(A.right,"i", '^');
                                infarray[max][1]=Parse(A.left,"i", '^');
                                max++;
                                break;
                        default: System.out.println("This is not a correct inference");
                                System.exit(0);
                }
}

//Works practicaly the same as and()
public static void or(argument A){
        switch(A.value){
                case "0": argument A1 = Parse(A.right, "0", '^');
                                solve(A1);
                                argument A2 = Parse(A.left, "0", '^');
                                solve(A2);
                                break;
                case "1": infarray[max][0]=Parse(A.right,"1", '^');
                                infarray[max][1]=Parse(A.left,"0", '^');
                                conflict[counter]='s';
                                max++;
                                infarray[max][0]=Parse(A.right,"1", '^');
                                infarray[max][1]=Parse(A.left,"1", '^');
                                max++;
                                infarray[max][0]=Parse(A.right,"1", '^');
                                infarray[max][1]=Parse(A.left,"i", '^');
                                max++;
                                infarray[max][0]=Parse(A.left,"1", '^');
                                infarray[max][1]=Parse(A.right,"0", '^');
                                max++;
                                infarray[max][0]=Parse(A.left,"1", '^');
                                infarray[max][1]=Parse(A.right,"i", '^');
                                max++;
                                break;
                case "i": infarray[max][0]=Parse(A.right,"i", '^');
                                infarray[max][1]=Parse(A.left,"0", '^');
                                conflict[counter]='s';
                                max++;
                                infarray[max][0]=Parse(A.right,"0", '^');
                                infarray[max][1]=Parse(A.left,"i", '^');
                                max++;
                                infarray[max][0]=Parse(A.right,"i", '^');
                                infarray[max][1]=Parse(A.left,"i", '^');
                                max++;
```

```
                        break;
                default: System.out.println("This is not a correct inference");
                        System.exit(0);
        }
}

//Creates two paths, one where the left side of the inference is i, and one where it is
    1
public static void inference(argument A){
        switch(A.value){
                case "0": if(A.left!=null){
                                infarray[max][0]=Parse(A.left,"i", '^');
                                infarray[max][1]=Parse(A.right,"0", '^');
                                conflict[counter]='s';
                                max++;
                                infarray[max][0]=Parse(A.left,"1", '^');
                                infarray[max][1]=Parse(A.right,"0", '^');
                                max++;
                        }else{
                                A=Parse(A.right,"0", '^');
                                solve(A);
                        }
                        break;
                default: System.out.println("This is not a correct inference");
                        System.exit(0);
        }
}

public static void negation(argument A){
        switch(A.value){
                case "0": A = Parse(A.right, "1", '^');
                        solve(A);
                        break;
                case "1": A = Parse(A.right, "0", '^');
                        solve(A);
                        break;
                case "i": A = Parse(A.right, "i", '^');
                        solve(A);
                        break;
                default: System.out.println("This is not a correct inference");
                        System.exit(0);
        }
}

//If there is an implication, it is stored
public static void implication(argument A){
        infaanw=1;
        imps[counter][cimps]=A;
        cimps++;
}

//Gives both sides the same value
public static void comma(argument A){
        argument A1=Parse(A.left,A.value,'^');
```

```
        solve(A1);
        argument A2=Parse(A.right,A.value,'^');
        solve(A2);
}

//Finds the right operatorfunction for the (sub)argument
public static void solve(argument A){
        switch(A.operator){
        case '^': and(A);
                        break;
        case ',': comma(A);
                        break;
        case 'V': or(A);
                        break;
        case '=': inference(A);
                        break;
        case '!': negation(A);
                        break;
        case '>': implication(A);
                        break;
        default: if(A.left.length()==1 && A.right==null){
                              valuate(A.left, A.value);
                        }else if(A.right.length()==1 && A.left==null){
                              valuate(A.right, A.value);
                        }else{
                              System.out.println("This is not a correct inference.");
                              System.exit(0);
                        }
        }
}

//If there is more than one possibility for the values of subarguments in an inference
    (e.g. (a^b)>c where (a^b) is 0), this function goes through all the options
public static char splitPaths(int mdif, int i){
        int maxnow=max;
        copy(mdif+1); //Copy the current values of atoms to the new subpath
        char conf='1';
        for(int x=1;x<=mdif;x++){
                counter=(max-x);
                conflict[counter]='0';
                for(int y=0;y<2;y++){
                        if(infarray[counter][y]!=null){
                                solve(infarray[counter][y]);
                                }
                }
                if(conflict[counter]=='s'){ //If subpaths are created in the subpaths,
                    the function goes through them recursively
                        int mdif2=max-maxnow;
                        conflict[counter]=splitPaths(mdif2,i);
                }
                if(conflict[counter]=='0'){conf='0';} //If there is at least one
                    possibility that the conflict of a path is zero, splitPaths returns
                    zero
        }
```

```
        return conf;
}

//Checks if the current set of answers is unique and if it is not an array filled with
    null-values
public static int uniqueAns(int e, String[][] ansar){
        int unique=0;
        int justnull=1;
        for(int x=0;x<9;x++){
                        if(ansar[e][x]!=null){
                                justnull=0;
                        }
                }
        if(e==0){       //For the first set of answers, it just checks if it isn't just
            null
                if(justnull==0){
                        return 1;
                }
                return 0;
        }
        for(int q=0;q<e;q++){
                for(int x=0;x<9;x++){
                         if(ansar[q][x]!=ansar[e][x]){ //If there is one value in a set
                             that differs from e, unique is incremented with 1
                           unique+=1;
                           x=9;
                        }
                }
        }
        if(justnull==0){
                if(unique>=e){ //If unique is as high or higher than the number of
                    answer-sets, the current answer-set is unique
                        return 1;
                }
                return 0;
        }else{
                return 0;
        }
}


//This function goes through all of the implications of all the subpaths and checks
    what values the left and right side of the implications can take, given the
    possible values of the sides of the previous implications
public static void results(){
        String ans[][][] = new String [max][10000][9];
        int cmax=max;
        int first=1;
        int count=0;
        int e=0;
        String[][] ansar= new String[10000][9];
        for(int i=0;i<cmax;i++){ //Goes through all the current paths
                int mnow=0;
                int ca=0;
```

```
counter=i;
//It copies the conflict en values of the atoms of the current i in max
    and makes max 1 higher
conflict[max]=conflict[i];
max++;
int m=max;
copy(2);
if(conflict[i]=='0'){
for(int j=0;j<cimps;j++){    //Goes through all the implications
       if(imps[i][j]!=null){
       mnow=max;
       for(int k=0;k<9;k++){ //Goes through every possible combination
           of values for the left and right side of the implication
              for(int l=m-1;l<mnow;l++){   //Compares these with every
                  possible combination of values for the previous
                  implications
                     if(conflict[l]=='0'){
                     for(int f=0;f<9;f++){ans[i][max][f]=ans[i][l][f];}
                         //Copies the current set of answers to max
                     char a='p';
                     char b='q';
                     max++;
                     counter=l;
                     copy(2);       //Copies the current values of atoms
                         to max-1
                     counter=(max-1);      //Counter is now max-1, which
                         has the values of the atoms and the set of
                         answers of l
                     conflict[counter]='0';
                     switch(k){     //Every value of k stands for a
                         combination of values
                            case 0:
                                          argument
                                              A=Parse(imps[i][j].left,"0",'^');
                                          solve(A);
                                          argument
                                              B=Parse(imps[i][j].right,"0",'^');
                                          solve(B);
                                          a='0';
                                          b='0';
                                          break;
                            case 1: A=Parse(imps[i][j].left,"0",'^');
                                          solve(A);
                                          B=Parse(imps[i][j].right,"1",'^');
                                          solve(B);
                                          a='0';
                                          b='1';
                                          break;
                            case 2: A=Parse(imps[i][j].left,"0",'^');
                                          solve(A);
                                          B=Parse(imps[i][j].right,"i",'^');
                                          solve(B);
                                          a='0';
                                          b='i';
```

```
                    break;
            case 3: A=Parse(imps[i][j].left,"1",'^');
                    solve(A);
                    B=Parse(imps[i][j].right,"0",'^');
                    solve(B);
                    a='1';
                    b='0';
                    break;
            case 4: A=Parse(imps[i][j].left,"1",'^');
                    solve(A);
                    B=Parse(imps[i][j].right,"1",'^');
                    solve(B);
                    a='1';
                    b='1';
                    break;
            case 5: A=Parse(imps[i][j].left,"1",'^');
                    solve(A);
                    B=Parse(imps[i][j].right,"i",'^');
                    solve(B);
                    a='1';
                    b='i';
                    break;
            case 6: A=Parse(imps[i][j].left,"i",'^');
                    solve(A);
                    B=Parse(imps[i][j].right,"0",'^');
                    solve(B);
                    a='i';
                    b='0';
                    break;
            case 7: A=Parse(imps[i][j].left,"i",'^');
                    solve(A);
                    B=Parse(imps[i][j].right,"1",'^');
                    solve(B);
                    a='i';
                    b='1';
                    break;
            case 8: A=Parse(imps[i][j].left,"i",'^');
                    solve(A);
                    B=Parse(imps[i][j].right,"i",'^');
                    solve(B);
                    a='i';
                    b='i';
                    break;
    }


    if(conflict[counter]=='s'){  //If the current
        combination of values causes new sub-paths to
        be made, the program checks what

                    values the atoms can take in
        those sub-paths
           int cnow=counter;
           int mdif=max-(counter+1);
```

```
        if(j==cimps-1){ //If this is the last
            implication, it suffices to see if any
            of the sub-paths is possible
               conflict[counter]=splitPaths(mdif,i);
               counter=cnow;
        }else{ //If this is not the last
            implication, the implications after this
            one have to be compared with all the
            sub-paths, so
                                you have to copy
            the parameters from the current l into
            the new sub-paths
               int u=splitPaths(mdif,i);

               counter=cnow;
               if(u=='0'){

                      //Copy the answers from the
                          current counter to all
                          the sub-paths
                      for(int r=1;r<=mdif;r++){
                             for(int d=0;d<9;d++){
                                    ans[i][max-r][d]=ans[i][counter]
                             }
                      }
                      String str=(a + ">" + b + "="
                          + imps[i][j].value);
                      if(ans[i][counter][k]==null){
                             //Put str in the
                                 answerset of all
                                 the sub-paths
                             for(int
                                 r=1;r<=mdif;r++){
                                    ans[i][max-r][k]=str;
                             }

                      }else
                          if(ans[i][counter][k].equals(str)){
                      }else{
                             //Make conflict 1 for
                                 all the sub-paths
                             for(int
                                 r=1;r<=mdif;r++){
                                    conflict[max-r]='1';
                             }
                      }
               }
        }
}else if(conflict[counter]=='1'){
}if(conflict[counter]=='0'){
        String str=(a + ">" + b + "=" +
            imps[i][j].value);
               //If there isn't an answer in this k
                   yet, put this one in
```

```
                                        if(ans[i][counter][k]==null){
                                                ans[i][counter][k]=str;
                                        //If there is already an answer but
                                            it's the same as the one you
                                            want to put in, do nothing
                                        }else
                                            if(ans[i][counter][k].equals(str)){
                                        //If there is already an answer and
                                            it's different, there is a
                                            conflict
                                        }else{conflict[counter]='1';}
                                }
                        }
                        }
                }m=mnow+1;
        }
                if(j==cimps-1){ //When you are at the last implication, put the
                    answer sets in a new array
                        mnow=max;
                        for(int l=m-1;l<mnow;l++){
                                if(conflict[l]=='0'){
                                        for(int x=0;x<9;x++){
                                                if(ans[i][l][x]!=null){
                                                        ansar[e][x]=ans[i][l][x];
                                                }
                                        }
                                        if(uniqueAns(e,ansar)==1){   //Check if the
                                            answer set is unique
                                                e++;
                                        }else{ //If it is not unique, clear it
                                                for(int q=0;q<9;q++){
                                                        ansar[e][q]=null;
                                                }
                                        }

                                }
                                conflict[l]='0';
                        }
                }
        }}
        max=cmax;
}
if(e==0){
        System.out.println("If you want this inference to be invalid, the
            resulting logic is not truth-functional");
}
for(int q=0;q<e;q++){ //Print or if this isn't the first set of answers that is
    printed
        if(q>0){
                System.out.println("or");
        }
        for(int r=0;r<9;r++){ //Print the answers
                if(ansar[q][r]!=null){
                        System.out.println(ansar[q][r]);
```

```
                        }
                    }
                }
            }

    public static void main(String args[])
    {
            InputStreamReader istream = new InputStreamReader(System.in) ;
            BufferedReader bufRead = new BufferedReader(istream) ;
                    String value = "1";
                    String inference = "a";
                    for(int i=0;i<26;i++){     //Makes the conflict for all the possible
                        paths zero
                           conflict[i]='0';
                    }
                    //Read the inference from line input
                    try {
                            System.out.println("Please Enter The Inference: ");
                            inference = bufRead.readLine();
                            }
                    catch (IOException err) {
                            System.out.println("Error reading line");
                    }
                    argument A = Parse(inference, "0", '='); //Parse the inference
                    infarray[0][0]=A;
                    //Solve the inference and all the resulting sub arguments
                    for(int i=0;i<max;i++){
                          counter=i;
                          int cmax=max-1;
                          for(int j=0;j<5;j++){
                                 if(infarray[i][j]!=null){
                                        solve(infarray[i][j]);
                                 }
                          }
                          //If new sub-paths are created, create them and copy the right
                              values of atoms
                          if(conflict[counter]=='s'){
                                 copy(max-cmax);
                                 copyImps(max-cmax);
                          }
                    }
                    if(infaanw==1){ System.out.println("\nRestrictions: \n"); //If there is
                        an implication in the inference, print out the restrictions
                    results();}
                    else{System.out.println("There is no implication in this inference.");}
                        //If there are no implications in the inference, print out that fact
    }
}
```

References

B. Kooi and A. Tamminga. Completeness via correspondence for extentions of the logic of paradox. *The Review of Symbolic Logic*, 5(4):720–730, 2012.

G. Priest. The logic of paradox. *Journal of Philosophical Logic*, 8:219–241, 1979.

G. Priest. *An Introduction to Non-Classical Logic.* Cambridge: Cambridge, 2001.

H Sahlqvist. Completeness and correspondence in the first- and second-order semantics for modal logic. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, volume 2, pages 110–143. Amsterdam: North-Holland Publishing Company, 1975.

J. van Benthem. Modal correspondence theory. *PhD Thesis*, 1976.