Obligationes

Making an interactive website around a medieval game.

Date: 27-6-2008

Authors:
Irma Cornelisse, 5619750 (irma.cornelisse@live.nl)
Patrick Mast, 0240866 (Patrick.Mast@gmx.de)
Ricus Smid, 9660208 (hesmid@chello.nl)
Djura Smits, 5619807 (djura_s@hotmail.com)

Supervisors:
Jaap Maat
Sara Uckelman

Contents

Introduction

This report will describe what we did to implement the game obligationes and to get it on the web. Before we can describe this, it has to be explained what the game obligationes actually is.

What is obligationes?

Obligationes (or "obligations") is a formal disputation form that was widespread in medieval Europe. The earliest writings on obligations date from the beginning of the thirteenth century, but the theoretical roots can probably be found much earlier, assumably in Aristotle's *"Topics"*. Obligationes can be viewed as a game between two players, the opponent (*opponens*) and the respondent (*respondens*). The opponent puts forward some hypothesis and the respondent decides whether he denies or admits the hypothesis. In the first case the game doesn't start, in the latter case the game is on its way. The opponent puts forward questions (propositions) that may or may not relate directly to the hypothesis. The respondent answers these questions with 'I concede', ' I deny' or 'I doubt it'. This is where the name of the game comes in: both players are *obliged* to follow a very strict set of rules that determine how a question should be answered according to both the hypothesis, the propositions already put forward and the real world. When the respondent follows these rules closely, he or she can maintain a consistent 'world' that follows logically from the original hypothesis. The goal for the opponent is to trick the respondent in 'responding badly' within the game time that the players agreed upon. When the game time is up or when the respondent has responded badly, the opponent ends the game by saying *"cedat tempus"*.

One interesting aspect of this logical game is that, while it is clear that obligationes were widespread and heavily discussed from the thirteenth century on, the actual purpose of the game remains unclear.

There are many versions of the game, but we will work on two versions: positio and dubitatio. In positio, the first fact (the hypothesis) has to be held true through the game. In dubitatio, the first fact has to be held in doubt through the game.

Our goal

Our goal is to implement positio and dubitatio, so that a player can act as the respondent, and the computer plays the opponent. This implementation, along with extra information about the game, has to be accessible via a website. Furthermore, other students should be able to implement their own versions of the different games and add it to the website.

Problems and Solutions

Implementation of positio

Introduction

Positio is the most common version of obligationes. In positio, the first statement given by the opponent should be held true throughout the game. The statements following should be answered so that it does not contradict the previous statements. When the previous statements do not influence the current question, the answer should be according to the real world.
The version of positio that we used was described by Walter Burley. We have encountered quite some problems while implementing the reasoning module.

Decision on rules and domain

The first problem was that we had to find out the rules of the obligationes game. We were given a number of articles that described the rules for the different versions. In many of these articles these rules get quite complex, mostly due to the many exceptions to the rules that arise in very specific, or less specific situations. We cannot possibly take all exceptions into account because the game would then get very frustrating for users who have not heard about obligationes before. So we had to distinguish between the main rules and the less important ones. Also, many writers have different views of the game and give a different set of rules, so we also had to choose a writer who had the set of rules that would make the game the most interesting and the least frustrating for new users.

The next thing we had to decide on, was what the domain was going to be. In the chosen domain, all facts should have the same truth value for every player, otherwise the player's mistake does not only lie in wrong reasoning, but also in having other ideas about the facts, such as "you are in Rome". So we have to find a domain that is correct for (almost) everyone. Choosing the right domain is also important for minimizing the number of exceptions that can occur.

By doing a lot of reading and discussion, we have gained insight in the different versions of obligationes and decided that we were going to use the basic rules for positio that Burley described. Burley also wrote a lot about specific exceptions, but we decided to only stick with the basic rules. Burley sums up all the basic rules, and explains all the exceptions that can occur after each rule. For positio there are just a few basic rules that enables the opponent to respond correctly if the domain is chosen rightly. We chose to start with a domain that consists of sentences like "you are in Rome", "you are a human", "grass is green", and "you are the pope". When using these sentences, we have a few assumptions about the user. Firstly, we assume that the user is sitting in front of a computer. Secondly, we assume that the user is not the pope. We also assume that the user is not in Rome while playing the game. Lastly, we have the trivial assumption that the user is human. We have chosen these sentences because most of these facts are also used in the examples written by the medieval authors, so it gives a more authentic atmosphere.

Interpreting logic

Another problem we encountered was that we had to find a way to decide if a formula is sequentially relevant, sequentially contradictory, or irrelevant. We converted all formulas to conjunctive normal form to reduce the complexity of the formulas. When we have done that, only three connectives remain, namely and, or, and negation. If we want to check if a formula is possible, we have to check if each element of the conjunction is possible. Checking the possibility of atomic formulas or negated atomic formulas is pretty straightforward. The disjunctions though, are more problematic. The disjunctions just say that at least one element is true, but it is not known which element. Therefore, we have to implement several rules for interpreting the disjunctions. The same problems with disjunctions occur when a formula has to be added to the knowledge base. For instance, when it is decided that a certain atomic formula, or the negation of an atomic formula is true, all disjunctions containing that element do not matter any more. Also, the negation of this formula has to be removed from all disjunctions. When a disjunction completely matches a disjunction in the list of ors, we should not add that new disjunction to the list, because it is already in there. This is just a simple form of pattern matching. However, it can also be the case that a disjunction is equivalent to a disjunction in the list, but the elements are not in the same order. We have to find a way to also match disjunctions if they are not exactly the same, but are equivalent.

We have solved the problem of interpreting disjunctions by keeping a separate list for disjunctions and one for atomic formulas and negations of atomic formulas. There is not much reasoning to do about the list without disjunctions, but after every step, the program has to check if elements of disjunctions in the list of disjunctions can be discarded or complete disjunctions can be discarded. When an atomic formula or the negation of an atomic formula is added to the list of truths, the reasoning module checks if the list of disjunctions contains a disjunction that contains this formula. If this is the case, the disjunction can be completely discarded. When a disjunction contains the negation of the formula, that element is discarded from the formula. When a “disjunction” only contains one element, because the other elements have been discarded, this element is moved to the list of truths. When an element is moved to the list of truths, there has to be another check to see if other disjunctions should be modified.

Updating the lists is done when the respondent has answered correctly. When the respondent correctly answered “I concede”, the conjunctive normal form of the posited fact is added to the knowledge base. When the answer was “I deny”, the same is done to the negation of the formula. Now, when a question is posited, and the respondent has answered, the reasoning module will first check if the posited fact is “possible”. In this case, we mean by possible that the fact is not contradictory with the knowledge base of the current game, not with the knowledge base of the real world. We check this by checking one by one if an element of the converted formula is not contradictory with the list of truths or disjunctions. When it is not contradictory, the element is temporarily added to the knowledge base, until all elements are checked. When it turns out that a fact is not possible, the program knows the given answer has to be “I deny”. The next thing that has to be checked is if a formula is irrelevant. This is done by taking each disjunction, atomic formula, and negation of an atomic formula of the formula and checking if the list of truths or disjuncions contains this formula. If one element of the conjunction is contained in the knowledge base of the game, the formula is relevant. Otherwise, it is irrelevant. When it is irrelevant, determining the answer is done by checking if the fact is possible but now using the

real world as knowledge base.

Making the module intelligent

The reasoning module has to be able to ask smart questions. For several cases there are strategies that have a high probability of trapping the opponent. For instance, when a respondent responds with "I doubt it", we know in most cases that the respondent has done badly. Often, the respondent thinks a fact is irrelevant while it is relevant or forgets that an element of the formula is true in the world model. There are many cases in which specific strategies do not apply; in those cases, a general strategy has to be applied that still gives a higher chance of answering wrongly. One important general strategy is using elements of the positio domain that have not been used in a long time. There is a high chance that the user has forgotten what kind of answer he has given to questions far in the past. We cannot use only this as a general strategy, because then at the beginning, only new domain elements will be introduced, because the distance between the first and the last answer is not yet big enough to ask the first questions. Therefore, newly introduced elements should start with a high chance because it is easier to trap an opponent if you combine recently introduced elements with new ones. In this way you also can force the respondent to deny or concede a fact that was not used before in the domain of the current game.

When these strategies do not apply to a certain case, the module has to use a general strategy so that the questions are not completely random, but still have a higher probability to be answered wrongly.

We have chosen a few strategies to enable the opponent to ask smarter questions. First of all, we implemented a general strategy that already resulted in pretty “intelligent” questions. We made a list that contained all elements that could be chosen to use in a question. To every element in that list we attach a value. At the beginning of a game, all values will be one. When an element is used for the first time, this value will be set to ten, or twenty when it is used in the positum. When such an element is used again, the value will be decreased with one. When an element has to be chosen, we create a more or less imaginary list, in which each element is repeated as many times as is indicated by the value. Then we choose a random number of which the maximum is the sum of all values. Then we choose the value that has this random number as index. At random times, we introduce a more specific strategy. This strategy forces the opponent to implicitly make one element of the positum true. Because the elements of the positum are not true in the real world, there is a chance that the respondent still thinks this element false and will answer wrongly when he is asked to concede or deny this fact. It may be clearer to give an example:

Opponent	**Respondent**	**Explanation**
I posit that grass is purple or the pope is a woman	*I admit*	Because this is the positum and is not contradictory, the respondent should admit this.
Grass is purple	*I deny*	This is irrelevant because the positum does not imply that this is true or false. This fact is untrue in the real world, so it should be denied. This kind of question is generated first by our first strategy

The pope is a woman	*I concede*	This kind of question is generated in the second stage of strategy 1.Because the first element of the disjunction of the positum is denied, the second element should be true, therefore, this should be admitted. Because this fact is untrue in the real world, people can be tempted to deny this fact.

This strategy can be applied only once in the game. We also implemented a strategy that can theoretically be applied infinite times, but in our game, it is limited by the number of facts we have in our domain. With this strategy, the opponent introduces a fact into the game that has not been mentioned in the game. This introduced fact is the opposite of a fact that is true in the real world. The following example will show how this strategy is used. This example continues the game of the previous example, so the game model (in which "the pope is a woman" is conceded) can be reused.

Opponent	**Respondent**	**Explanation**
The pope is not a woman or you are running	*I concede*	This statement is irrelevant, because the first element of the disjunction is not true in the game model, and the second fact does not yet have a truth value in the game model because it has not been mentioned before. This would be the first step of our second strategy.
you are running	*I concede*	This statement has to be conceded because that is the only way in which the previous sentence can be true in the game model. This is a very tricky strategy. Now that the previous statement has been introduced into the game model, the real world values do not count anymore, and answers have to be given according to completely different facts. This question is generated by the second stage of strategy number 2.

Furthermore, we give the user the choice to play in easy or in difficult mode. Difficult means that the maximum length of the formulas is higher. This not necessarily makes the program more intelligent, but then it makes better use of it's ability to process difficult formulas in the correct way.

Natural language

Another set of problems will appear when we want to convert the logical formulas to natural language. The elements of the formulas cannot be directly mapped to words or a sequence of words because then neg(p) will be converted to "not the pope is in Rome", instead of "the pope is not in Rome", which is what we want to say. Another problem that occurs is that when elements are directly mapped, the resulting sentences are ambiguous. (p\/q)/\r will map to the same sentence as p\/(q/\r), for instance: "the pope is in Rome or you are a man and you are running". We have to find a way to make the parentheses explicit in natural language, and maybe emphasize certain parts of the sentence. Solving this problem is crucial for playing obligationes, because the user has to know what exactly is meant by the question, because there is exactly one right answer for each question.

We chose to make the conversion from formulas to natural language not too complex. The elements of the formulas are more or less directly converted to a sequence of words in natural language. There do occur some problems when we do this. As is said before, something like neg p cannot be directly converted to “not the pope is in Rome”, but has to be converted to “the pope is not in Rome”. To do this, we simply created a predicate for each atomic formula that converts it to its corresponding words, and did the same for the negations of these formulas. Now we still have the problem that sentences can be ambiguous. To solve this, we substituted every right parenthesis with a comma. This way, we can distinguish between (a\/b) /\ c and a\/(b/\c), because now the corresponding sentences are (for instance) “You are in Rome or you are running, and you are the pope.” and “You are in Rome, or you are running and you are the pope.” . Because natural language is not as exact as logical formulas, it would be convenient to give a few examples of games before a beginner starts the game, so he or she realizes the difference between these two sentences.

Ending the game

It seems somewhat simple, making the game shout “cedat tempus” at the right time, when deciding if a formula is possible and irrelevant has already been implemented. Yet this gave us more problems than we expected. Because we were working in prolog, the program tries to backtrack every time it fails, and this caused the game to shout “cedat tempus” multiple times in a row, and continued the game when it had to be ended. We also had to be sure that our code took care of all possible combinations of types of facts (sequentially relevant, sequentially contradictory etc) and answers (doubt, concede and deny). And we had to make sure that a game really ends after the game has said “cedat tempus”. To be able to do this, we first have to be sure that we are aware of all different cases of combinations of answers and relevance, possibility, etc of the generated facts.

The first case in which the user can give the “wrong” answer is when the user denies the positum. This part is easy, we simply linked the ‘I deny’ button to the first game page. Because it is not entirely wrong to deny the positum, we did not display “cedat tempus”.

The very first thing we check when the given fact is not the positum, is if the user has given a different answer before to the same question. To check if the user has answered wrongly we first look at the possibility of the fact in the game domain. Depending on the possibility, we do different further checks. There is only one wrong case for when the fact is not possible in the game domain, this is when the user does not deny this fact. There are more different possibilities when the fact is possible in the game domain. It can be that the fact is relevant and true, but it can also be that the fact is irrelevant, in the last case there are still three possible answers to give. For the first case, we only have to check if the answer was “I concede”. For the second case we do the same possibility check as before, but now on the real world domain. We also check if the fact is doubtful.

Explaining the user’s mistakes

In order to make the game understandable to a wider public, it is necessary to be able to explain to the user what he did wrong. The ideal explanation would be one of the form pointing out which rule should be followed and why it should be followed. So the game has to point out whether the user has previously answered the question differently, or whether the user was inconsistent, or, if the question was irrelevant, point this out and say what the right answer would be according to the real world domain, or, if the question was relevant, point this out and say which earlier question made this question relevant and whether it should be conceded or denied (so if it was logically relevant or incompatibly relevant).

Unfortunately, this is very hard to do for our implementation of the game, because we do not use the posited sentences to check what the answer should be. Instead, we convert all questions to conjunctive normal form and add the elements of that conjunction to the knowledge base. When it is added, various elements in that knowledge base will be changed or even thrown away because they may not be necessary any more. In this way, we cannot point out which previous questions have influenced the truth value of the current answer. This really was a problem because an explanation really is desirable for getting familiar with how the game works, but the way we have implemented the game seemed totally unusable for this kind of explanation.

We foresaw too much problems for creating an explaining module as we actually wanted it, so we decided to use our energy on a much simpler and straightforward way to implement an explanation module. This module would only output the direct consequences from your answer to a specific question given the earlier answers and the positum. This was how the program held the new information. Though it seems like very good help (rather than an explanation) it was too much information so it was more confusing then helpful. Now it became clear that an explanation would only be helpful if the program only gives information that is relevant. Anyway an explanation to the user what he had done when he answered badly really was desirable. Hence we decided to attempt to make a better explanation, namely one that would be much more like the explanation we wanted in the first place. We made some adjustments to the original code so that we knew when the user had answered badly, what the right answer had to be and whether the question was relevant or irrelevant or the mistake was that previously a different answer was given to the same question. So when a bad answer was detected by the program it holds that information. With this information the program ‘explain.pl’ can do his work. When two different answers were given we just look in the answer-lists (there are three of them: list with every question conceded in the game, list with every question denied in the game and a list with every question doubted in the game) to find in which list the question can be found. If the question was irrelevant we just say it was irrelevant and give the right answer (which is the information in our world model). If the question was relevant, we say it was relevant and give the questions and whether they were conceded or denied or whether it was the positum, which made the question relevant. Because we know that it was relevant and what the right answer would be, we only have to check with all the questions asked earlier to see which combination makes the question relevant. We do this by making a conjunctive normal form of the question and look for every or-part whether we can find an earlier question which makes this part relevant and otherwise find a combination of earlier questions which makes that part relevant. If we find relevant questions for every or-part of the conjunction, the whole conjunction is relevant because of the conjunction of these relevant questions together.

Implementation of Dubitatio

Introduction

Dubitatio is sometimes seen as a variation of *positio*. S. L. Uckelman, J. Maat and K Rybalko[1] argue that dubitatio is an art in itself, and not just a trivial variant of positio. In this game the aim for the respondent is to uphold the *dubitatum* as doubtful. The propositions put forward by the opponent should be answered with either concede, deny or doubt, as is the case in positio. Medieval literature is less informative about dubitatio then it is about positio.

It seems that there was no real consensus about the rules of the game: rules that are described in *Obligationes Parisienes* are different from the rules described by other authors like Nicholas of Paris and Pseudo-Sherwood.

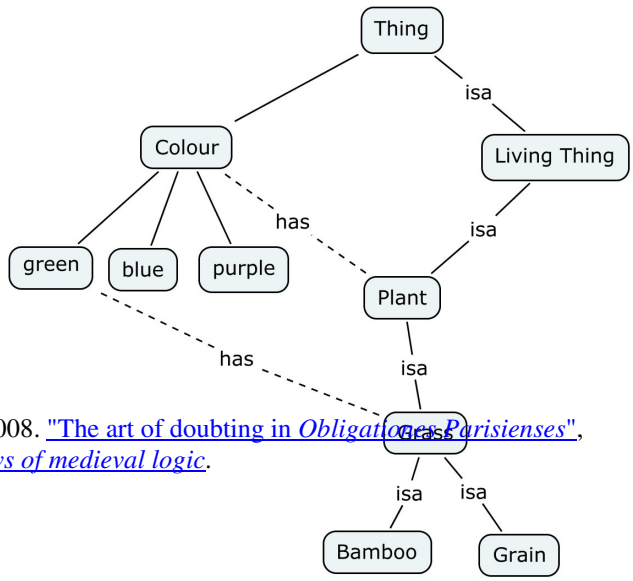
Approach

We decided to implement dubitation from scratch, instead of using the already built implementation for positio. There were two reasons for this: first of all, we wanted to make a world model that in principle was suited to support adding and removing concepts to and from the world in a meaningful way. This would provide future users with the possibility to customize the world to their own needs. Furthermore it would support adding new facts during the game or adjusting the world model as a consequence of new facts (e.g. 'you are in New York'). Secondly we thought it was interesting to provide the website with two different implementations, so that future programmers can draw inspiration from multiple approaches. This also is in line with the modular setup of the website, which encourages enhancement and improvement by future programmers and makes it possible to link different implementations of obligationes to the site.

We did come as far as a 'beta version' of the game, that works reasonably well, but still has some bugs and flaws. Also the game is not integrated as part of the website yet.

World Model

We implemented the world model as a semantic network consisting of a hierarchical tree, describing relations between different concepts. The 'isa' relation describes that some concept is 'a kind of' some other concept. The root of the tree is the concept 'thing', so everything in our world 'is a thing'. The 'has-a' relation



[1] Uckelman, Sara L.; Maat, Jaap; and Katherina Rybalko. 2008. "The art of doubting in *Obligationes Parisienses*", submitted for inclusion in proceedings volume *Modern views of medieval logic*.

describes that some concept 'has' some other concept, e.g. the concept *grass* 'has' the concept *green*. The program can reason about relations, e.g. it can infer that bamboo is coloured since bamboo is a grass and grass has the colour green and green is a colour.

fig. A simple world model

Creating the dubitatum and propositions

We started with building a kind of random generator to create a dubitatum. The dubitatum thus created consisted of a disjunction of two propositions that described any kind of existing relation in our world. To make the dubitatum more interesting we then decided to only use concepts that were a living thing (humans, plants) in the first part of every proposition followed by either a has-a relation with an attribute concept (e.g. colour) or an isa relation with some other living thing either higher up in the tree or in a different branch of the tree (e.g. the pope is-a tree). Also a negation was sometimes added to a proposition. A randomly generated dubitatum could look like this: 'the pope is an oak or not all plants are blue'.

Language model

To actually get the output described above, a simple language model was also needed. Logic formulas needed to be translated to natural language, concepts had to be provided with the right determiners and attributes had to be preceded by the right verbs. Furthermore we had to handle formulas like *neg entry(plant, colour, green)* (we used the 'entry' predicate to describe relations between concepts), which could mean 'all plants are not green' or 'not all plants are green'.

Implementing the rules of *dubitatio*

We decided to use the rules provided by the *Obligationes Parisienes*, an early 13th century medieval treatise about obligations. The formalized rules look like this:

1. $\Phi(d) = D$ (definitional).

2. if $p \rightarrow d$, then $\Phi(d) \neq C$

3. if $d \rightarrow p$, then $\Phi(d) \neq N$

4. if $d \leftrightarrow p$, then $\Phi(d) = D$

5. if $p = \neg d$, then $\Phi(d) = D$

6. if $p \rightarrow d$, if $V(p) = F$, if $\Phi(\neg p) \neq D$, then $\Phi(p) = N$

7. if $d \rightarrow p$ and $V(p) = T$, then $\Phi(p) = C$

8. if $d \rightarrow p$ and $V(p) = F$, then $\Phi(p) = D$

9. if $p \perp d$ (read 'p is repugnant to d'), then it is one of these:

    - if contradictory per se, then $\Phi(p) = D$

- if otherwise contradictory or contrary, then

a) if V(p) = F, then $\Phi(p) = N$

b) if V(p) = T, then $\Phi(d) = D$

With Φ a function that takes as argument a statement and returns a reaction (an answer). The reaction can be either D (doubt), C (concede) or N (negate/deny). V is a function that maps a proposition on a truth value.

A close look at the rules reveals that some rules are specifications of other rules, e.g. rule 6 is a specification of the more general rule 2.

Therefore we decided to create our own set of rules for use in our program.

1. if $p \rightarrow d$,
   a) if V(p) = F, if $\Phi(\neg p) \neq D$, then $\Phi(p) = N$
   b) if V(p) = T, then $\Phi(p) = D$ [1]
2. if $d \rightarrow p$,
   a) if V(p) = T, then $\Phi(p) = C$
   b) if V(p) = F, then $\Phi(p) = D$
3. if $d \leftrightarrow p$, then $\Phi(d) = D$
4. if $p = \neg d$, then $\Phi(d) = D$
5. if $p \perp d$,
   a) if V(p) = F, then $\Phi(p) = N$
   b) if V(p) = T, then $\Phi(d) = D$

Applying the rules

An interesting difference with positio is that the respondent should reason about rule 1: is the proposition antecedent to the dubitatum? This can make the task for the respondent rather difficult, since the dubitatum could result from the proposition after many in-between reasoning steps. For our implementation the following difficulties arose: how much reasoning can we expect from the person playing the game and according to which world rules do we expect the person to reason? Is he or she supposed to know that bamboo is always green because grass is green, and is it in fact true (dead grass isn't green)? To handle this, we decided to provide the user with a world model that is assumed to be true (before the dubitatum is posited).

Then we had to decide how to handle the consequences of putting forward a dubitatum in our world model. We needed some way to reflect the notion of 'doubtful' in our world. We approached this problem by assuming that the dubitatum (D) should be regarded as not true until it is either proven or disproven. Since in our programming language Prolog something is true or not true, and never doubtful (closed world model), 'not true' means in effect false.

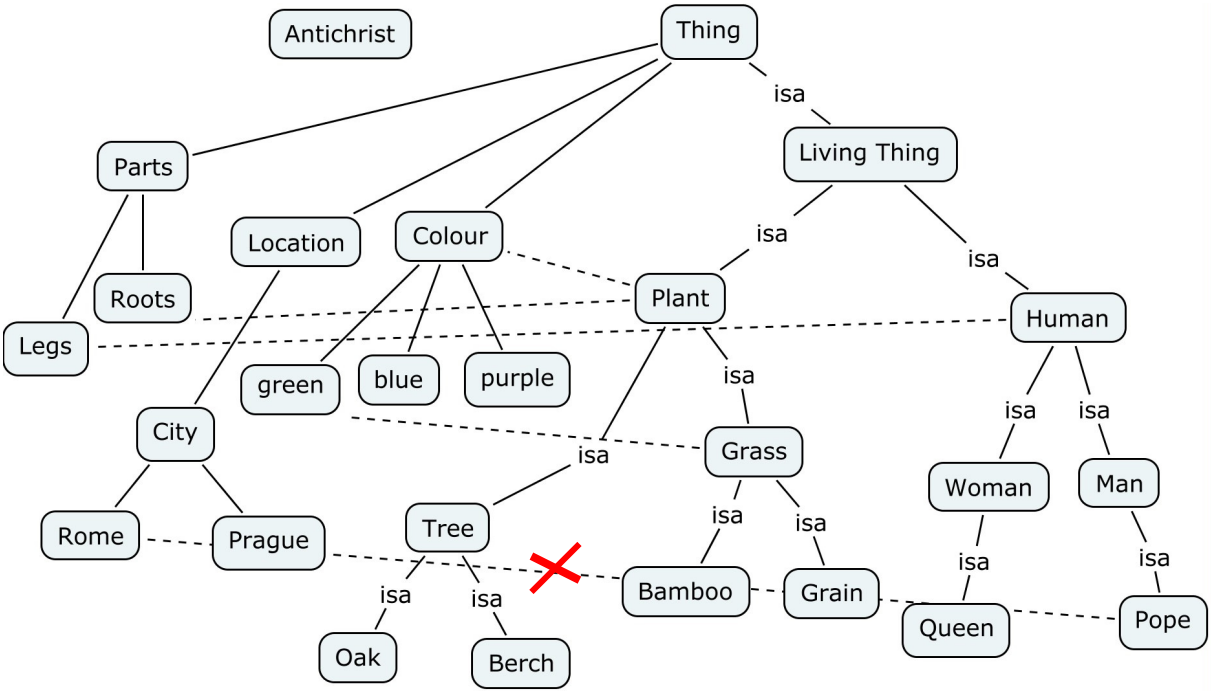
[1] We added this subrule ourselves, because it seems to be forgotten in the original rule set.

Therefore we decided to remove all facts from our world model that were equivalent to the dubitatum. We decided also to do some reasoning in advance when updating the world with the proposed dubitatum. While maintaining the original world model (RW = real world) in our program, we created a new world (DW = dubitatum world), such that also every fact antecedent or consequent to the dubitatum was removed. Of course, this world remains hidden to the respondent during the game.

Now, when a new proposition is put forward, the different rules should be applied to our world model(s). An example:
Rule 1 ($p \rightarrow d$) is applied as follows:

- The truth value of proposition p is determined in the DW. Since the dubitatum is always false in DW, $p \rightarrow d$ can only be true if p = F.
- If this is the case, then the proposition is made true, by adding concepts and or relations to the DW. So for the proposition '*a and b*' both *a* and *b* are made true in the DW, which leads to a third, temporary, world: TW.
- The truth value of $p \rightarrow d$ is determined in the TW, which is the same as determining the truth value of *d* (since we made *p* true).
- If d = T, then we know that *d* is a consequence of *p*.
- Finally we have to decide which sub rule of rule 1 applies, by determining the truth value V(*p*) of the proposition *p* in the real world.

An illustrated example:
Assume the Dubitatum: *The antichrist is purple or the pope is in Rome*
Only the second part is true in our world, so we have to remove the has-link between pope and Rome and we are done.

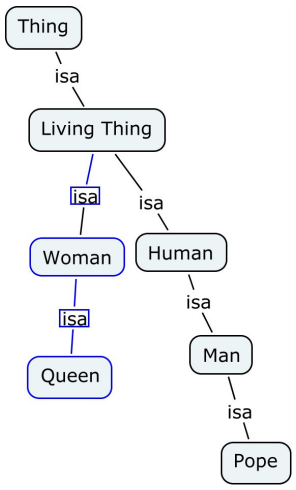


Suppose the following proposition is put forward: *'Every man lives in rome'*

Since this is not true in the DW, we create TW by adding it to DW. Then we check if the dubitatum is true in TW. This is the case (since every man lives in rome and since the pope is a man), so now we have to check the truth value of p in RW, which is false. By rule 1^a we have to answer the proposition with 'deny'.

Reasoning problems:

The example given above requires the program to maintain (backup and restore) three worlds, updating them with facts and restructuring them accordingly (including removing redundant facts, or adding new facts as a result of the update) and reasoning about the truth of propositions according to certain world relations and rules (e.g. if someone is a man, then he is not a woman). Also it should be able to add 'neg facts' to the database: facts that are not true. This is not the same as removing a fact, because in our model a fact that is not in the database may still be doubted, while a neg-fact is explicitly not true. Furthermore, reasoning can become quite complicated when 'hard facts' about the world are being doubted, as can be illustrated with the following example:
Suppose the dubitatum is: *'A woman is not a human'*. We can now create the DW as follows: Remove the entry *'woman is a human'* and add the entry *'woman is a living thing'* This way we keep intact as much as possible from the real world: we don't doubt the fact that a woman is a living thing.



Another way to look at it, however, is that a woman inherits 'being a living thing' from the fact that a human is a living thing. So why should we assume that a woman is a living thing, while doubting that a woman is a human? We could adopt the view that everything is a thing, and that a woman thus should be placed under thing, since that is all we know for sure.

We actually implemented this kind of reasoning, which becomes more complicated when putting forward subsequent propositions like 'the queen is a human'.

However, we decided in the end that the game might become unplayable with this kind of reasoning and all the (arbitrary) rules that should be provided to the player. We adopted the view that we should hold on to the is-a relations from the real world, since those are often the more 'hard facts'. We decided to allow mainly has-a relations to be doubted in the dubitatum, which makes the game far better understandable.

Playing the game

To actually play the game, the program first creates a dubitatum and then opens the dialogue with the player. When the player answers 'doubt' the game starts and the program creates a propostion. We added the possibility to play the game on three different levels (easy, medium

and hard) which results in propositions of different complexity. The program determines the answer the player should give by applying the rules. When no rule can be applied, it is assumed that the proposition is irrelevant[1] and the answer is determined according to the truth value of the proposition in the real world. The resulting answer is compared with the answer given by the player. If the answers are different, the game ends ('cedat tempus') and the player is informed what the answer should have been and what rule he or she should have applied.

Technical problems

On the technical level we encountered different problems. We decided to use Prolog for the programming part. Prolog is a programming language that is especially suitable for logic problems. But Prolog is a quite specialized language and does not have any native interfaces that would have allowed us to use it as an Internet-scripting language.
One of our goals was exactly that: implementing the logic games and making them accessible via the website. Therefore we had to find a way to link Prolog to a website. We started off with using Python. We wrote a simple Python script that called Prolog with the necessary commands in an input file and then passes the results back to Python. Python is quite common in interactive web-programming and we thought we were on the right track.

As Prolog is non a standard programming language and is even less common on the web we had some troubles with bringing the website online and working. Our first intention was to use our student's web-space which is easily accessible via the *public_html* directory. The web-space is hosted on the same university servers as the student's home directory and so we knew that Prolog would be available. But, due to higher security policies, Python scripting is disabled on the student accounts, and even worse: writing files to the *public_html* directory with a web-script was disabled as well. So we had no web-space for the interactive web-site and we had no idea how to find a server with Prolog running.

After having talked to our supervisors, we got in contact with the ILLC's webmaster and we were allowed to host our website on their servers. There we had sufficient rights to run the website and a (quite old) version of Prolog was available, too.
Python scripting was still a problem. The university discourages its use. Therefore we decided to switch to PHP. PHP is developed for web-scripting but does have less common programming capabilities. But we were still able to use it. After having rewritten all necessary scripts, we had the website up and running.

Normally, when a program is run in Prolog, the program is executed in one single environment. All temporal data is stored within this environment. In our case, the game progress and the game settings were stored temporarily. With web-scripting a program is restarted every time the user gives new input. This behavior is contrary to the Prolog behavior and therefore we had to adjust our program. The temporary data had to be stored in a file so that the user can continue his game every time again.

[1] Whether or not something is relevant in relation to dubitatio is not really explained in the texts we have seen so far.

The moment we published the URL to our website and several users played the game at a time, we discovered a major drawback of our way to link Prolog and PHP and the way we make Prolog usable with PHP. As we save important game settings in a file and we only used one file, the settings from different players constantly got mixed up. For the user this resulted in unexplainable behavior and wrong results. We finally fixed this problem by giving every user an individual file. From that moment on it was possible for the website to handle several users at a time.

When we first had the website up and running and we started to play the game on our own we discovered some features we missed. We definitely missed the possibility for the website to give a reasonable explanation the moment the user did something wrong. We also missed the options to show the logical formula and to determine the game level, we thought it would be handy to show the history so that the users could trace back their line of reasoning, and we absolutely missed a high-score list to add an inter-user-competition.

The positio code takes advantage of some shortcuts in reasoning (rewriting implications to disjunctions, mixing propositions with the world model,...) and therefore it was not possible to properly analyze why the user behaved badly. To get nice explanations, we had to make major adjustments to the original code and we had to write an explanation module. The implementation of the explanation part is explained elsewhere in the paper.
On the other hand, showing the logical formula and determining the game level were already implemented in the program and were quite easily added to the website. We only had to add the option to show this part of the Prolog result.

Prolog saves all earlier propositions, but in a simplified form. To provide the user with a real history, we decided to handle the history within the PHP scripts. We had to save the previous propositions, the answers to that propositions and (if requested) the logical formula.

How the website works

Our website is online at: http://www.illc.uva.nl/medlogic/obligationes . When you visit this page, you will see an introduction to obligationes, and buttons on the left. Most buttons are not clickable, but show what further work can be done. The intention of this is that the buttons can be made clickable when other students have implemented their own versions and put it on the site. The following buttons are currently clickable:

Home

When you click on home, the website will show the first page again. This page shows some general information about the game obligationes, but does not describe a version in particular.

Play the game

This is the most important part. When clicking this button, two frames will open instead of one. This is the first page of the positio game. In the left frame the game options can be selected and the game can be started. The right frame gives information about the left, so here it gives information about what happens when different options are chosen. We will not explain the first

two options (*Show history while playing* and *Show logical formula*) because they are explained on the left of the website. It is also possible to choose if the game is difficult or easy (easy is default). Easy means that the formulas will not contain more than two binary connectives. When the user chooses difficult, the formulas can contain up to four binary connectives. It is advisable to check the *show logical formula* checkbox when playing the difficult game, because the longer sentences are very difficult to interpret in natural language. It is also possible to directly view the high score list. There is a separate list for easy and difficult mode. When the user clicks play the game, a different page will open. The left page now shows a randomly generated positum, while the right page shows information about what will happen when clicking the different buttons. A user should always admit the sentence if he wants to continue the game. When *deny* is clicked, the previous page will open again. The only reason to click deny is if the player does not want to play with the given positum. When admit is clicked, a similar page will open. This time, a fact is generated to which the user can answer *I concede*, *I deny*, or *I doubt it.* In the frame on the left is given a reminder when the player should answer with which answer. If the user has given the right answer, another similar page with a different proposition will open. This goes on until the user has given the wrong answer. When that is the case, *cedat tempus* will appear in red letters. Below that, the user can give his or her name and submit it so it can be displayed on the high score list if the score was high enough. The user can also click the *why?* button and see why he has answered wrongly. The score is displayed below this button. The last button can be clicked if the user wants to play the game again. This was the most important part of the website.

Positio

Positio gives a short introduction to the game positio. This introduction is focused on the elements of the positio game that are relevant to the game on the website. It also gives two examples of important and tricky parts of the game. These examples describe the questions that are generated by the two strategies that are implemented in the game.

Dubitatio

This part of the site gives a short introduction to the dubitatio game. It also describes the rules that have to be used when playing dubitatio. The dubitatio game itself is unfortunately not online (yet).

About this site

About this site gives some general information about why this website is made. It shows the people who made the game and the website and the people who have supervised this project.

Further readings

This page gives a very short list of literature for people who want to know more about obligationes.

Sourcecode

This page gives all the source code of the different prolog programs that are used. It also contains

a link to the documentation of the prolog code. This page is for other students who may want to extend the website.

Statistics

The small button below the menu gives access to information about how many people have visited this website.

Conclusions and discussion

Positio

The implementation of positio has given great results. We are very glad it works on the website after several implementing and also practical difficulties. Implementing the rules of the game was seriously a lot of work because of the great difficulty of the medieval game. The rules had a lot of consequences which were hard to foresee. We had to make sure the game would still be interesting to play, despite the existence of strategies which would be fatal for the user.

It was a pity we had to keep the domain small and make sure we didn't overlook any implicit inference as decrypted in the report. However the game positio and nice extras such as high-score and an explanation module are available on the internet. So we suppose we can say this part of the website is totally as we hoped it would be.

Dubitatio

As mentioned in the previous section, the game is not quite finished yet, although we think it is pretty close. The main flaws at the moment are:

- The random creation of propositions is really random. We have not added any strategy yet. This may be solved by using the same strategy as is used in *positio*.
- The rules turn out to be not entirely correct. For example the rule p → neg d is missing from the rules provided by *Obligationes Parisienes*, while it should obviously be there. This rule still should be added in our program.
- The game should actually not end until the respondent is forced to either deny or concede the dubitatum (or when he or she answers badly to an irrelevant proposition).
  This means the program should recognize and remember the moment at which the respondent gives an answer that eventually will lead to his loss. The world model should be then updated with the facts that follow from this incorrect answer. Furthermore the program should be able to lead the respondent quickly to the end of the game, and explain exactly where he or she lost the game.
  This seems a rather complex task (although we haven't given it a try yet). It may be quite difficult to find a good strategy to lead someone to his or her loss quickly. Also explaining what went wrong, beyond 'you should have applied that rule to that proposition', seems not something very easily done.

Further possibilities

We are very content with everything we have reached in four weeks time, but of course if more time was given to us there are still a lot of other things that could have been done to make the website more complete.

When visiting the site one of the first things one would notice is that there are missing links behind depositio, institutio, rei veritas, petitio. We never meant to fill these pages with information and games, but the goal was to make an interactive website about obligations and the idea behind the empty links is that in some other project these links can be filled with information. Of course it would be real nice if for all the variants of obligations there would be a possibility to play the game. Unfortunately we see some big problems arise because the information for positio was already hard to get on one line (get one rule set) because of all the different interpretation from writers about this game. For dubitatio it was even harder to get an understandable interpretation of the rule set. So we expect a lot of problems when one would implement one of the other versions of obligatio. Our experience when we read about obligations was that there was very little information about other variants than positio. As we find it very hard to get the challenge of the game positio (because we could not find that anywhere in the literature), we foresee problems by finding challenge in the other variants and ask ourselves if there is even enough information to know what can be posited and what has to be answered in all possible cases (even in dubitation we could not really find what to do with propositions which are irrelevant, so one case wasn't covered, which is very hard to deal with if you are working with computers).

A more promising game to implement would be a different variant of positio and then we think about the variant of Roger Swyneshed, which actually should be very straight forward to implement with the code written to play positio in Burley's manner. But here it would be a challenge to find out what is the difficulty in this game and to get the program to play an interesting game with the user.

At the beginning of the project we also had the idea that it would be very nice that two people could play the game together over internet and the program could be of help by checking if everything said is according to the rules. So the computer will then be a sort of referee during the game. We think that would be a very nice option.

In the line of the previous point it could also be nice that the program could give hints during the game to the user, like some practising mode. Then users first could get familiar with the game before playing the game for 'real'.

Our implementation of positio is using a fixed world model. This world model of course can be adjusted. This means a bigger and more complex world model is possible. Something that doesn't work actually as we want it to work is the possibilities to give answers to questions. We have chosen for propositions about which we could make a good approximation whether they would be true, false or doubtful for the user who is playing the game. But of course you could think of propositions from which we don't know whether they are true, false or doubtful for the person who is playing the game (let's say: 'it is raining'). And it is also possible for things that we supposed to be doubtful are known by the player of the game (for example: 'it is raining in Prague' maybe the user is in Prague, or is listening to the weather report). So the thing is it

would be nice for the program to accept for some (!) propositions that the user answers true or doubtful and that it accepts for some (!) other propositions that the users answer may be false or doubt. We think it would be still a lot of work to decide what should be the answer and which answer we definitely are sure to be false. But as it works now there are some improvements to make at this point.

Appendix A: setup of the website

The folder that contains the PHP and Prolog files should contain a subfolder "files". This folder is used from different scripts to store temporal data. This folder should have "`reading`", "`writing`" and "`execution`" rights for "`others`". The `.htaccess` file is used to secure this folder from the execution of malware and should be placed in the folder.

The file `prolog.php` contains the function that calls Prolog. This function should get the actual path to the Prolog binary. The variable `$prologPath` should be set to the correct value (example: `$prologPath = "/opt/pl-4.0.9/bin/pl";`)

The file reasoningExplain.pl is the main reasoning Prolog program. The temporal data is stored to a file. The path to the file and the file name are specified with the predicates `facts_file` and `facts_path` the path should point to the directory files mentioned above (example: `facts_file('facts.pl').` and `facts_path('files/').`).

File listing:

`about.html`	contains the "bout" part of the website
`choiceRules.html`	contains the help given on the "play the game" start page
`dubitatio.html`	contains the a short explanation on dubitatio
`further_readings.html`	contains the "further_reading" section of the website
`game-frameset.html`	the frameset that shows the for the game necessary pages: help pages and the main game pages
`gameRules1.html`	contains the help given on the "positum"-game-page
`gameRules2.html`	contains the help given on the "proposition"-game-pages
`header.html`	contains the header/title part of the website
`index.html`	the starting page, shows the menu, the header page and the game-frameset
`menu.html`	contains the navigation menu of the website
`positio.html`	contains the a short explanation on positio
`sourcecode.html`	contains the a source code part of the website
`welcome.html`	contains the welcome screen
`delete_file.php`	script that deletes the temporary files at the end of the game
`game1.php`	the first page of the game: shows the positum
`game2.php`	processes the answer on the positum and shows the first proposition
`game3.php`	processes the answer to a proposition and shows the new proposition
`gameChoices.php`	the menu of the game, initializes the session and starts the game
`highscore.php`	shows the high score lists
`prolog.php`	script to link prolog to PHP
`explain.pl`	the Prolog file that gives the explanation on an user error
`reasoningExplain.pl`	THE Prolog files, does the whole reasoning for the positio game
`upgrade.pl`	a small helper file to make our prolog file compatible with the older (4.0.9)

	prolog version we found on the server
worldModel.pl	the Prolog file that contains the world model representation of the positio game
body.css	contains the style sheet for different parts of the website
homebody.css	contains the style sheet for different parts of the website
menu.css	contains the style sheet for different parts of the website
optionbody.css	contains the style sheet for different parts of the website
style.css	contains the style sheet for different parts of the website

Directory: files

.htaccess	file to prevent the Server from executing scripts in this folder and thus to prevent mallware to be run
difficulthighscore.txt	conmtains the high score for the difficult game mode
easyhighscore.txt	conmtains the high score for the easy game mode

Directory: images (contains all the images of the website)

button_aboutthissite.jpg
button_depositio.jpg
button_dubitatio.jpg
button_furtherreadings.jpg
button_home.jpg
button_institutio.jpg
button_petitio.jpg
button_play.jpg
button_positio.jpg
button_reiveritas.jpg
button_sourcecode.jpg
cedat_tempus.gif
difficultbutton.jpg
easybutton.jpg
embrace_small.jpg
highscore.gif
playagainbutton.jpg
rand.gif
startbutton.jpg
title.jpg
whybutton.jpg

Directory: prolog_doc (contains the documentation files)

explain.html
index.html
reasoningExplain.html

Appendix B:

usage of the Prolog program positio

Start prolog.

`consult('reasoningConsult.pl').`
Load the positio program

`play.`
Start the game and generate the positum (all temporal files are made).

`positio1(answer, level).`
Answers to the positum and returns a new proposition. The answer possibilities are: 'admit' or 'deny', the level can be set to 'easy' or 'difficult'

`positio2(answer, level).`
Answers to a proposition. Returns a new proposition. Returns 'cedat tempus' if the user made an error. The answer possibilities are: 'concede', 'deny' or 'doubt', the level can be set to 'easy' or 'difficult'

`explain.`
Gives an explanation after the user got an 'cedat tempus'

usage of the Prolog program debutatio

Start prolog.

`consult('dubitatio.pl').`
Load the dubitatio program

`go2.`
Start a 'scenario' with the dubitatum and the first few propositions fixed. The following propositions are randomly created.

`rules.`
`Shows rules that are applied, while playing the game`
`norules.`
`No rules are showed.`