

# Satisfaction algorithm

Wouter Beek & Remko Scha

November 25, 2008

## 1 Introduction

In a natural language interface we want to ask questions to the system and then get a full and correct answer to them. Since we already know how to analyze questions in a logical language, and since we have computational procedures for handling logical expressions, we can now use those procedures in order to generate answers to natural language questions.

In the case of a yes/no-question, for example “Does John walk?”, we want to know whether John walks or not, i.e. we want to know what the truth-value of  $\|Walk(John)\|_{\mathcal{M}}$  is. We already know how to translate natural language assertion sentences into their logical equivalent. In the case of “John walks.” we get  $Walk(John)$ , or  $Assert[Walk(John)]$  in order to indicate that the logical expression is to be used as an assertion. In a similar vein “Does John walk?” translates into  $Query[Walk(John)]$ .

The speech act operators ‘*Assert*’ and ‘*Query*’ indicate how we are to make use of their argument expression. Such speech act operators can therefore be thought of as invoking pragmatic operations: If a natural language sentence  $S$  is analyzed as  $Assert[\phi]$ , then we add  $\phi$  to our knowledge-base. If a natural language sentence  $S$  is analyzed as  $Query[\phi]$ , then we give the appropriate answer  $A$ , relative to the knowledge that is stored in our knowledge-base. Our knowledge-base therefore takes on the function of the logical model  $\mathcal{M}$  with respect to which the logical expressions are to be evaluated.

## 2 Satisfaction: answering yes/no-questions

We will now provide a mechanism for handling yes/no-questions. This means that if a sentence  $S$  is a yes/no-question, to be analyzed as  $Query[\phi]$ , our answering mechanism will yield ‘yes’ in case  $\|\phi\|_{\mathcal{M}} = 1$  and ‘no’ in case  $\|\phi\|_{\mathcal{M}} = 0$ .

The mechanism that does this is called ‘satisfaction’. A model  $\mathcal{M}$  satisfies a sentence  $\phi$  iff  $\mathcal{M} \models \phi$  iff  $\|\phi\|_{\mathcal{M}} = 1$ . Once we have a successful satisfaction algorithm establishing the truth value of logical sentences, the

last step is to output ‘yes’ in case this algorithm succeeds for the given formula, and outputs ‘no’ in case the algorithm does not succeed.

Let us therefore mechanize the process of satisfaction first, by giving a Prolog-predicate `satisfy`. In doing this we will be using the principle of polarity, according to which proving that a sentence holds is the same thing as proving that its negation does not hold (and vice versa). The mode, either the satisfiability of a sentence  $\phi$  or the satisfiability of its negation, is kept track of using the fourth argument of our Prolog-predicate `satisfy`. The first argument of the predicate is the sentence to be evaluated (i.e.  $\phi$ ), the second argument is the model, and the third argument is the set of relevant substitutions we have made.

The models are taken to have structure `model(N, D, I)`. The `N` is the name of the model, so that we can have an arbitrary number of models in our Prolog knowledge-base. So for example  $\|\phi\|_{\mathcal{M}}$  differs from  $\|\phi\|_{\mathcal{N}}$ . We will use this at a later point, when we are modelling belief spaces. For the moment we will just make use of the standard model  $\mathcal{M}$ , whose name is `[m]`. The `D` in the model-structure is the domain of the model, and the `I` in the model-structure is the interpretation function of the model.

An example of a model is:

```
model([m], [d1, d2, d3, d4, d5], [
    f(0, jules, d1),
    f(0, vincent, d2),
    f(0, pumpkin, d3),
    f(0, honey_bunny, d4),
    f(0, yolanda, d5),
    f(1, customer, [d1, d2, d4, d5]),
    f(1, walk, [d3, d5]),
    f(2, love, [(d1, d2), (d2, d3)])]).
```

It is named `[m]` (the standard model). Its domain consists of 5 entities. Its interpretation function gives constants for all entities, and defines three unary predicates and one binary predicate. So everyone except for Pumpkin is a customer, Pumpkin and Yolanda walk, and Jules loves Vincent and Vincent loves Pumpkin. For the binary predicate definition the ordering of the sublists is important, so Jules loves Vincent but not the other way round.

We start out with the satisfaction algorithm for atomic formulas; this makes use of the Prolog-predicate `i` that evaluates the terms that make up the atomic formulas by using the interpretation function of the model. The structure of the interpretation function of the model is, as we have seen, a list of structured occurrences of the form `f(Arity, Constant, Value)`, where `Arity` is the number of arguments the predicate denoted by `Constant` takes (zero for individual constants). `Value` consists of sequences of the indicated arity for whom the predicate holds. We give the clauses for satisfying atomic

formulas consisting of unary predicates here, those for predicates of higher arities can be added in a straightforward way:

```
satisfy(A, model(_N, D, I), G, pos):-
    A =.. [P | [Arg]],
    i(Arg, model(D, I), G, V),
    member(f(1, P, Vs), I),
    member(V, Vs),
    writef('%w is satisfied with G is %w\n', [A, G]).
satisfy(A, model(_N, D, I), G, neg):-
    A =.. [P | [Arg]],
    i(Arg, model(D, I), G, V),
    member(f(1, P, Vs), I),
    \+ member(V, Vs),
    writef('neg %w is satisfied with G is %w\n', [A, G]).
```

The interpretation function either looks for a variable's denotation in the assignment function, or it looks for the denotation of an individual constant. Here the structure of the assignment function is relevant; it consists of structured occurrences  $g(X, V)$ , where  $X$  is the variable to be replaced and  $V$  is the individual constant replacing it. We cannot represent a complete assignment function, since this is always an infinite construct assigning values to the infinite set of variables in the logical language. We therefore only represent the assignments  $g(X, V)$  that are relevant for the satisfaction of the expression at hand, and don't care about the other assignments:

```
i(X, model(_D, I), G, V):-
    (var(X), member(g(Y, V), G), Y==X, !)
    ;
    (atom(X), member(f(0, X, V), I)).
```

We now extend the satisfaction predicate to cover complex formulas, built out of the atomic ones by truth-functional operations. Negation is defined by appeal to the polarity argument. For conjunction and disjunction we use the Prolog-operators `,` and `;` respectively. Negated conjunction is defined in terms of disjunction and vice versa. The implication is defined in terms of disjunction and negation. The bi-implication is defined in terms of conjunction and implication:

```
satisfy(not(A), M, G, pos):-
    satisfy(A, M, G, neg).
satisfy(not(A), M, G, neg):-
    satisfy(A, M, G, pos).
satisfy(and(A, B), M, G, pos):-
    satisfy(A, M, G, pos),
    satisfy(B, M, G, pos).
satisfy(and(A, B), M, G, neg):-
```

```

        satisfy(A, M, G, neg)
        ;
        satisfy(B, M, G, neg).
satisfy(or(A, B), M, G, pos):-
    satisfy(A, M, G, pos)
    ;
    satisfy(B, M, G, pos).
satisfy(or(A, B), M, G, neg):-
    satisfy(A, M, G, neg),
    satisfy(B, M, G, neg).
satisfy(imp(A, B), M, G, Pol):-
    satisfy(or(not(A), B), M, G, Pol).
satisfy(bi(A, B), M, G, Pol):-
    satisfy(and(imp(A, B), imp(B, A)), M, G, Pol).

```

The predicate for quantification is slightly more tricky. Evidently we can define universal quantification in terms of existential quantification and negation. To check for the satisfiability of an existentially quantified formula is just to check for a suitable candidate in the domain that can do the job:

```

satisfy(exists(X, A), model(N, D, I), G, pos):-
    member(V, D),
    satisfy(A, model(N, D, I), [g(X,V) | G], pos).
satisfy(exists(X, A), model(N, D, I), G, neg):-
    setof(
        V,
        (member(V, D),
         satisfy(A, model(N, D, I), [g(X,V)|G], neg)
        ),
        D
    ).
satisfy(forall(X, A), M, G, Pol):-
    satisfy(not(exists(X, A)), M, G, Pol).
satisfy(forall(X, A), M, G, Pol):-
    satisfy(not(exists(X, A)), M, G, Pol).
satisfy(forall(X, A), M, G, Pol):-
    satisfy(not(exists(X, A)), M, G, Pol).

```

The hardened Prolog-programmer will recognize a problem here. Prolog will take different occurrences of the symbol **X** to be the same variable, whereas in the logical notation this need not be the case, e.g. `exists(X, and(walk(X), exists(X, talk(X))))` will result in a failure just in case there is no one who both walks and talks, whereas  $\exists x(Walk(x), \exists(x, Talk(x)))$  also holds if John walks and Peter talks. The problem is easily circumvented by requiring uniqueness of variables in the program's input, e.g. `exists(X,`

`and(walk(X), exists(Y, talk(Y)))` for the above case, although a nice predicate that does this for you is preferred.

**Exercise 2.1.** Implement the satisfaction algorithm in Prolog. Create a number of test expressions that show that the algorithm is valid, and make sure that all of the intermediary  $\beta$ -reduction steps are shown in the program's output.

**Exercise 2.2.** Add your implementation of the satisfaction algorithm to your Prolog-program for evaluating natural language sentences, so that when a user poses a natural language yes/no-question, your Prolog-program gives the right answer.

### 3 Answering wh-questions

As with yes/no-questions, wh-questions will be analyzed using the ‘*Query*’ speech act operator. So a wh-question  $S$  will be analyzed as  $Query[\phi]$ . Our current Prolog code will therefore try to evaluate whether a model  $\mathcal{M}$  satisfies  $\phi$ , i.e.  $\mathcal{M} \models \phi$ . But the semantics for Wh-questions is of the form  $\lambda x.(\dots x \dots)$ , and our current satisfaction-algorithm cannot handle lambda's.

In a question/answer dialogue system we would however like to give an answer to wh-questions, namely the enumeration of entities  $x$  for which the expression  $(\dots x \dots)$  yields 1. Let us first give an example of this. We again use the model:

```
model([m], [d1, d2, d3, d4, d5], [
    f(0, jules, d1),
    f(0, vincent, d2),
    f(0, pumpkin, d3),
    f(0, honey_bunny, d4),
    f(0, yolanda, d5),
    f(1, customer, [d1, d2, d4, d5]),
    f(1, walk, [d3, d5]),
    f(2, love, [(d1, d2), (d2, d3)])]).
```

The wh-question “Who walks?” is evaluated as  $Query[\lambda x.Walk(x)]$ . We want the answer to this question to be “Pumpkin and Yolanda”. We know that an entity in the domain can be substituted for  $x$  in  $Walk(x)$  iff  $\exists x.Walk(x)$  is true. By successively invoking the satisfaction-predicate for  $\exists x.Walk(x)$ , we can therefore extract all  $d_i \in D$  for which  $\lambda x.Walk(x)$  yields 1. We then gather all these  $d_i$  in a list in order to get the answer we are looking for. The Prolog code for this is as follows:

```
bagof(V,
    (member(V, D),
```

```
satisfy (Formula, model(N, D, I), [g(X,V) | G], pos)),
VV)
```

For the above example we would get as an answer  $VV = [d3, d5]$ . This is not yet our projected answer “Pumpkin and Yolanda.”, but since  $d3$  and ‘Pumpkin’, and  $d5$  and ‘Yolanda’, are linked in the interpretation function  $I$ , this last step should be trivial.

There is one more complication though. Just in case there is no  $d_i \in D$  such that  $Walk(x)[d_i/x]$  is true, the above piece of Prolog-code would yield **false**. But we don’t want our answer to ‘fail’ (whatever that may mean), instead we want to answer with the empty list. E.g. for an alternative model:

```
model([n], [d1], [
    f(0, jules, d1),
    f(1, walk, [])]).
```

we want to answer “Nobody” to the question “Who walks?”. Again, the distance between answering  $[]$  and answering “Nobody” is taken to be trivial. The following piece of Prolog-code will solve for this ‘Nobody’-case:

```
\+ satisfy (exists(X, Formula), model(N, D, I), G, pos),
VV = []
```

Putting these two pieces together results in:

```
satisfy (lam(X, Formula), model(N, D, I), G, pos):-
    ((\+ satisfy (exists(X, Formula), model(N, D, I), G, pos),
    VV = []))
    ;
    bagof(V,
        (member(V, D),
         satisfy (Formula, model(N, D, I), [g(X,V) | G], pos)),
    VV),
    writef('The answer for %w is %w.\n', [X, VV])).
```

Please remark that this does not work for answering questions involving tuples as answers, as in “Who loves who?”, though it does of course answer “Who does John love?”. Nor does this algorithm work for answering questions involving higher order entities, e.g. as in “What do John and Peter have in common?”

**Exercise 3.1.** Implement the Prolog-clause for answering Wh-questions. Create a test instance that shows that the algorithm is valid, and make sure that all of the intermediary  $\beta$ -reduction steps are shown in the program’s output.