Fixed-Point Semantics for Non-Monotonic Formalisms

P. Rondogiannis
(joint work with (i) W.W. Wadge (ii) V. Kountouriotis and Ch. Nomikos)

FICS 2010, August 21-22, 2010

Outline

Outline

Logic Programming:

One of the most important applications of Mathematical Logic in Computer Science.

Example

```
ancestor(X,Y) ← parent(X,Y).
ancestor(X,Y) ← parent(X,Z),ancestor(Z,Y).
parent(john,nick).
parent(nick,tom).
```

Semantics of Logic Programming:

It was first developed in [van Emden and Kowalski, JACM, 1976]. The idea can be explained by a simple example:

Example

```
study ← rains.
play ← sunny.
sunny.
```

The program has three models:

$$
\begin{aligned}
M_0 &= \{(\texttt{sunny}, \texttt{True}), (\texttt{play}, \texttt{True}), (\texttt{rains}, \texttt{False}), (\texttt{study}, \texttt{False})\} \\
M_1 &= \{(\texttt{sunny}, \texttt{True}), (\texttt{play}, \texttt{True}), (\texttt{rains}, \texttt{False}), (\texttt{study}, \texttt{True})\} \\
M_2 &= \{(\texttt{sunny}, \texttt{True}), (\texttt{play}, \texttt{True}), (\texttt{rains}, \texttt{True}), (\texttt{study}, \texttt{True})\}
\end{aligned}
$$

Model M_0 is the right one from a computational point of view (it does not contain atoms that are true for an arbitrary reason).

Theorem:

Every logic program P (that does not use negation in clause bodies) has a minimum model M_P.

What does minimum mean in this context?

If N is any other model of P then the set of true atoms of M_P is a subset of the set of true atoms of N.

Example

Consider the models of the program of the previous example:

$$\begin{aligned} M_0 &= \{(\texttt{sunny}, \texttt{True}), (\texttt{play}, \texttt{True}), (\texttt{rains}, \texttt{False}), (\texttt{study}, \texttt{False})\} \\ M_1 &= \{(\texttt{sunny}, \texttt{True}), (\texttt{play}, \texttt{True}), (\texttt{rains}, \texttt{False}), (\texttt{study}, \texttt{True})\} \\ M_2 &= \{(\texttt{sunny}, \texttt{True}), (\texttt{play}, \texttt{True}), (\texttt{rains}, \texttt{True}), (\texttt{study}, \texttt{True})\} \end{aligned}$$

M_0 is obviously the minimum model of this program.

Two basic characteristics of logic programming:

- Every program has a unique minimum model.
- We can find this model by just looking at the set of models of the program, without caring about the syntax of the program.

Actually, this model can be computed as the least-fixed point of a continuous operator associated with the program.

Logic programming supports a form of negation known as "negation-as-failure".

Intuitively:

The query $\sim A$ succeeds iff our attempt to prove A terminates and fails.

Example

```
works ← ~ sleeps.
sleeps.
talks ← ~ studies.
```

According to negation-as-failure, `works` should be taken as false because sleeps is true and `talks` should be taken as true because we can not prove the truth of `studies`.

The concept of negation-as-failure has very important practical applications.

Example

In a data base for a university department there exists a relation `enrolled(Student,Course)`. If we do not use negation-as-failure, we must also have a relation `not-enrolled(Student,Course)`. Relations of this kind may be huge (without conveying essential information).

Main applications:

Logic Programming, Data Bases, Artificial Intelligence.

Semantic approaches for negation-as-failure:

- *Stable model semantics [Gelfond and Lifschitz, 1988].* The semantics of the program is defined through a syntactic transformation (*stability transformation*). Every program can have zero or more stable models.
- *Well-founded semantics [van Gelder, Ross and Schlipf, 1991].* It uses a logic based on three truth values (True, 0 and False). It can be proved that every logic program with negation has a distinguished *well-founded* model.

In this talk we will focus on the second approach.

Example

Consider the program:

```
works ← ~ tired.
tired ← ~ sleeps.
sleeps.
```

The well-founded model of the program is:

$$M = \{(\texttt{sleeps}, \mathit{True}), (\texttt{tired}, \mathit{False}), (\texttt{works}, \mathit{True})\}$$

The well-founded model is usually constructed based on the syntax of the program. The program is partitioned into strata according to the dependencies through negation, and the computation of the model is performed starting from the lower strata and moving towards the upper ones.

Example

Consider the program:

$$\boxed{\mathtt{p} \leftarrow \sim \mathtt{p}.}$$

The well-founded model of the program is:

$$M = \{(\mathtt{p}, 0)\}$$

The program can not be partitioned into strata. The value `0` assigned to `p` has the meaning "I can not decide if `p` is `True` or `False`".

An annoying observation:

The well-founded approach is not purely model theoretic.

Example

Consider the program:

$$\texttt{works} \leftarrow \sim \texttt{tired}.$$

The models of the program are:

$$\begin{aligned} M_0 &= \{(\texttt{tired}, \mathsf{False}), (\texttt{works}, \mathsf{True})\} \\ M_1 &= \{(\texttt{tired}, \mathsf{True}), (\texttt{works}, \mathsf{False})\} \\ M_2 &= \{(\texttt{tired}, \mathsf{True}), (\texttt{works}, \mathsf{True})\} \end{aligned}$$

There is no minimum model! The models M_0 and M_1 are minimal.
The well-founded model is M_0.

Example

Consider the program:

$$\texttt{tired} \leftarrow \sim \texttt{works}.$$

The models of the program are:

$$\begin{aligned} M_0 &= \{(\texttt{tired}, \mathsf{False}), (\texttt{works}, \mathsf{True})\} \\ M_1 &= \{(\texttt{tired}, \mathsf{True}), (\texttt{works}, \mathsf{False})\} \\ M_2 &= \{(\texttt{tired}, \mathsf{True}), (\texttt{works}, \mathsf{True})\} \end{aligned}$$

This program has the same set of models as the previous one.
However, its well-founded model is M_1.

Some remarks:

- There is no way to choose the correct model of a program by just looking at its set of (classical) models.
- For a given program, there does not exist in general a unique *minimum* well-founded model (but instead a set of minimal models out of which we must somehow choose the well-founded model).
- Does this imply that classical logic is not the right approach to the semantics of negation-as-failure?

Example

Consider the program:

```
works ← ~ sleeps.
sleeps.
talks ← ~ studies.
```

In the well-founded model the atoms `sleeps` and `talks` are both true. However, `sleeps` seems to be "truer" than `talks` (because there is a fact that asserts beyond any doubt that `sleeps` is true, while `talks` is true just because there is no indication that `studies` is true).

The above seem to imply that we need different levels of True and False values:

$$\mathsf{F}_0 < \mathsf{F}_1 < \mathsf{F}_2 \cdots < \mathsf{0} < \cdots < \mathsf{T}_2 < \mathsf{T}_1 < \mathsf{T}_0$$

Example

The program:

$$\begin{array}{l} \texttt{works} \leftarrow \sim \texttt{sleeps}. \\ \texttt{sleeps}. \\ \texttt{talks} \leftarrow \sim \texttt{studies}. \end{array}$$

has as "special" model the following:

$$M = \{(\texttt{sleeps}, \mathsf{T}_0), (\texttt{studies}, \mathsf{F}_0), (\texttt{talks}, \mathsf{T}_1), (\texttt{works}, \mathsf{F}_1)\}$$

Definition:

An interpretation I of a program P is a function from the set of atoms of P to the infinite set of truth values $\{F_0, F_1, \ldots, 0, \ldots, T_1, T_0\}$.

Definition:

Let I be an interpretation of P. We extend I as follows:

- For every literal $\sim p$:

$$I(\sim p) = \begin{cases} T_{i+1} & \text{if } I(p) = F_i \\ F_{i+1} & \text{if } I(p) = T_i \\ 0 & \text{if } I(p) = 0 \end{cases}$$

- For every conjunction of literals:

$$I(l_1, \ldots, l_n) = \min\{I(l_1), \ldots, I(l_n)\}$$

Definition:

Let P be a program and I an interpretation of P. We will say that I *satisfies* a rule of P of the form $\mathsf{p} \leftarrow \mathsf{l}_1, \ldots, \mathsf{l}_n$ if $\mathrm{I}(\mathsf{p}) \geq \mathrm{I}(\mathsf{l}_1, \ldots, \mathsf{l}_n)$. Moreover, I is a *model* of P if I satisfies all the rules of P.

Does this approach solve the problems we have mentioned with respect to the semantics of negation? One can show the following [Rondogiannis and Wadge, TOCL, 2005]:

Theorem:

Every normal logic program P has a unique *minimum* infinite-valued model M_{P}.

Theorem:

Let N_{P} be the interpretation that results if we replace all the T_i values in M_{P} with *True* and all the F_i values with *False*. Then, N_{P} is the well-founded model of P.

How do we compare two models in our logic?

"All truth values are equal but some truth values are more equal than the others".

In comparing two models:

- We first compare the sets corresponding to the zero-order values of the two models.
- If the two sets are equal, then we proceed to compare the next level of truth values.
- If on the other hand one of the two sets is better at the current level of comparison, we announce the corresponding model as the winner.

Remark:

We do not examine the values beyond the point where the two interpretations become different! In other words, the higher order values are considered *negligible* compared to the lower order values.

Example

Consider again the programs:

$$\boxed{\texttt{works} \leftarrow \sim \texttt{tired}.}$$

and

$$\boxed{\texttt{tired} \leftarrow \sim \texttt{works}.}$$

According to the new approach, the "right" model for the first is:

$$M_{P_1} = \{(\texttt{tired}, F_0), (\texttt{works}, T_1)\}$$

while for the second:

$$M_{P_2} = \{(\texttt{tired}, T_1), (\texttt{works}, F_0)\}$$

M_{P_1} is not a model of P_2 and M_{P_2} is not a model of P_1. The two programs do not have the same sets of models any more!

The minimum infinite-valued model can be computed as the least fixed-point of an operator associated with the program.

Intuitively:

- We start with the "empty interpretation" (all atoms have value F_0).
- We use the rules of the program to compute new values for all atoms.
- We repeat this process until the set of atoms that have level 0 values (namely F_0 and T_0) converges.
- We reset the values of the remaining atoms to F_1 and we start again the iterations.
- At the end of this process, all the atoms that have not received a value, take the value 0.

Outline

Formal Language Theory:

Important connections with programming languages and compilers.

A Success Story:

Context-Free Grammars and the syntax analysis phase of compilers.

Observation:

Context-Free Grammars (implicitly) use the logical operation of *disjunction* by allowing multiple rules for the same non-terminal.

Example

$$S \rightarrow aSb \mid \epsilon$$

(Almost) Obvious Question:

How about the other two logical operations (*conjunction* and *negation*)?

Context-free languages are not closed under intersection and complementation.

Consequences:

- We can not express: "A string is produced by this *and* that".
- We can not express: "A string is produced by this *but not* by that".
- Simple languages such as $\{a^n b^n c^n : n \geq 0\}$ and $\{ww : w \in \{a, b\}^*\}$ are not expressible.

In 2004, A. Okhotin introduced a (seemingly) innocent generalization of context-free grammars.

Definition

A Boolean grammar is a quadruple $G = (\Sigma, N, P, S)$, where Σ and N are disjoint finite nonempty sets of terminal and nonterminal symbols respectively, P is a finite set of rules, each of the form

$$A \to \alpha_1 \& \cdots \& \alpha_m \& \neg\beta_1 \& \cdots \& \neg\beta_n \qquad (m + n \geq 1, \alpha_i, \beta_j \in (\Sigma \cup N)^*),$$

and $S \in N$ is the start symbol of the grammar.

Usefulness:

Specifying the syntax of programming languages (and especially aspects that can not be expressed by context-free rules).

Example

The language $\{a^n b^n c^n : n \geq 0\}$ is generated by the grammar:

$$\begin{aligned} S &\rightarrow AB\&DC \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow bBc \mid \epsilon \\ C &\rightarrow cC \mid \epsilon \\ D &\rightarrow aDb \mid \epsilon \end{aligned}$$

Conjunctive Grammars:

Boolean grammars that (as in the above example) do not use negation.

Example

The language $\{a^m b^n c^n : m, n \geq 0, m \neq n\}$ is generated by the grammar:

$$\begin{aligned} S &\rightarrow AB\&\neg DC \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow bBc \mid \epsilon \\ C &\rightarrow cC \mid \epsilon \\ D &\rightarrow aDb \mid \epsilon \end{aligned}$$

Inclusions on Language Classes:

$$\text{CONTEXT-FREE} \subset \text{CONJUNCTIVE} \subseteq \text{BOOLEAN} \subseteq \text{DTIME}(n^3)$$

The semantics of Boolean grammars can not (at least trivially) be based on derivations (as is the case for context-free grammars).

Example

Consider the grammar $S \to \neg S$. Derivation semantics applied naively, does not seem to work:

$$S \implies \neg S \implies \neg\neg S \implies \cdots$$

Even a model-theoretic approach does not seem trivial. For the above grammar, this would mean finding a language L such that $L = \neg L$.

Boolean grammars as Logic Programs:

The syntax of Boolean grammars is very close to the syntax of Logic Programs. Negation corresponds to "negation-as-failure".

Well-Founded Semantics for Boolean Grammars [Kountouriotis, Nomikos, Rondogiannis, 2009]:

- It is based on *three-valued formal language theory* (it can be easily extended to infinite-valued formal language theory).
- It is the correct approach to the semantics of Boolean Grammars.
- It gives an $O(n^3)$ parsing algorithm for all Boolean Grammars.

Three-Valued Formal Language Theory:

The membership of a string w in a language L can have three possible values: *true*, *false* or *unknown*.

Example

The language that corresponds to the grammar $S \to \neg S$ is the one which assigns to every string in Σ^* the truth value *unknown*.

Theorem

Every Boolean Grammar has a unique well-founded model.

Again, the model can be computed using fixed-point techniques that mimic the ones for logic programs with negation.

Natural Questions:

What kind of languages are representable by Boolean Grammars?
What kind of languages are not representable?

It seems that questions of the above type are difficult to be answered in full generality (at least with our present knowledge).

How can we proceed?

- Start with conjunctive grammars (they are simpler).
- Start with the simplest possible alphabet (namely $\Sigma = \{a\}$).

Interesting Question [Okhotin, 2004]: Are the languages produced by conjunctive grammars over $\Sigma = \{a\}$ always regular?

- CFGs over $\Sigma = \{a\}$: only regular languages

Theorem (Jeż, 2007)

The set of equations:

$$\begin{cases} X_1 & \to & X_1X_3 \& X_2X_2 \mid a \\ X_2 & \to & X_1X_1 \& X_6X_2 \mid a^2 \\ X_3 & \to & X_1X_2 \& X_6X_6 \mid a^3 \\ X_6 & \to & X_1X_2 \& X_3X_3 \end{cases}$$

has the least solution $X_k = \{ a^{k \cdot 4^n} \mid n \geqslant 0 \}$, *for* $k = 1, 2, 3, 6$.

Strengthening of the above result [Jeż, Okhotin, 2007]:

For every recursively enumerable set X of natural numbers we can construct a conjunctive grammar over $\Sigma = \{a\}$ that produces a language that grows faster than X.

Despite these positive results, no results saying that some particular (simple) set *cannot* be represented by Boolean Grammars could so far be obtained.

The only known restriction is the $\mathsf{DTIME}(n^3)$ complexity upper bound for Boolean Grammars.

No techniques of proving non-representability of sets by Boolean Grammars are presently known.

Remark:

The Pumping Lemma fails even for conjunctive grammars.

Outline

Other uses of the infinite-valued approach?

Another nice application of the technique has been obtained to give semantics to disjunctive logic programs with negation [Cabalar, Pearce, Rondogiannis, Wadge, 2007]. Other applications?

Does it hold that CONJUNCTIVE $\subset$ BOOLEAN?

It has been suggested by A. Okhotin that a language that can separate these two is $\{ww \mid w \in \{a, b\}^*\}$.

Find a simple language that is not CONJUNCTIVE/BOOLEAN.

Possibly $\{a^{n^2} \mid n \geq 0\}$ or $\{a^{2^{2^n}} \mid n \geq 0\}$ is not Boolean [Okhotin, 2006].

The talk was based on the following articles:

- P. Rondogiannis and W. W. Wadge, Minimum Model Semantics for Logic Programs with Negation-as-Failure, ACM Transactions on Computational Logic, 6(2): 441-467, 2005.
- V. Kountouriotis, Ch. Nomikos and P. Rondogiannis, Well-Founded Semantics for Boolean Grammars, Information and Computation, 207(9): 945-967, 2009.

Thank you!