A three-valued semantics for logic programmers

Lee Naish
Computer Science and
Software Engineering
University of Melbourne

Outline

Motivation and Background

Three-valued semantics:

- procedural
- fixed point
- model theoretic

Verification

Conclusion

Why bother with semantics?

- Philosophy
- Language design
- Implementation
- Programming
- Verification
- Debugging

What question are we answering?

What is the meaning of this program?

Does this program mean what I intend?

Debugging

This work came out of work on declarative debugging

Possibly surprising since those who work on semantics consider weird useless programs to be "interesting" rather than buggy

But often its during debugging that we think most about the relationship between our code, its behaviour and our intentions (ie, programming language semantics)

Previous debugging work classified (sub)computations as being correct, erroneous or "inadmissible"

Verification

Our technical results show that your logic program has no wrong answers and no missing answers (in "all solutions" computations which terminate normally) if

- Each ground atom is considered true, false or inadmissible,
- Each true atom is the head of a clause instance with a true body (all conjuncts are true), and
- For each false atom, all matching ground clause instances have false bodies (at least one conjunct is false)

Assumes negated calls are ground when they are selected

Traditional semantics: Model theory

(Simple case without negation)

Interpretations map each ground atom to $true$ or $false$ (typically we refer to the set of true atoms)

Classical truth tables are used for conjunction and disjunction

M is a model if for every clause instance
$H \leftarrow B$, if B is in M then H is in M (no clause instance $\mathbf{F} \leftarrow \mathbf{T}$)

The intersection of two models is a model, so a least model exists, which is the least Herbrand model

Traditional semantics: fixed points

The immediate consequence operator:

$T_P(M) = \{H | H \leftarrow B_1, B_2, \ldots B_N$ is a ground clause instance and $\{B_1, B_2, \ldots B_N\} \subseteq M\}$

T_P is monotonic; a least fixed point exists — start with ϕ and apply T_P until fixed point is reached

Useful for program analysis and alternative "bottom up" evaluation strategies

Traditional semantics: Procedural

SLD resolution gives a "top down" goal directed evaluation strategy (top-down search of an SLD tree)

The *success set*, SS, is the set of ground atoms with successful derivations

The least model, least fixed point and success set all coincide!

Soundness: If the interpretation M is a model then any successful atom is in M

Beautiful, except...

... intended interpretations are typically *not* models!

```
% merge(As, Bs, Cs): Cs is the sorted list of numbers
% which is the multiset union of sorted lists As and Bs
merge([], Bs, Bs).
merge(A.As, [], A.As).
merge(A.As, B.Bs, A.Cs) :- A =< B, merge(As, B.Bs, Cs).
merge(A.As, B.Bs, B.Cs) :- A > B, merge(A.As, Bs, Cs).
```

Programmers find such a comment adequate for describing or specifying the behaviour (perhaps we should add that `As` and `Bs` are "input")

Do programmers treat such a specification as an interpretation and just create a program for which it is a model?

No!

Intended interpretations versus models

It *is* a model for the definition `merge(As, Bs, Cs) :- fail.`

Should it be the minimum model?

`merge/3` succeeds with non-lists, eg `merge([],a,a)`, though not `merge(a,[],a)`

Similarly, it can succeed with lists which are not sorted

Swapping the first two arguments in all `merge/3` atoms in the clause heads, and/or just one/both of the recursive call(s) results in different success sets

Which of these 8 different versions is correct according to the (single) intended interpretation?

Which of `merge([2,3], [2,1], [2,1,2,3])` and `merge([2,3], [2,1], [2,2,1,3])` are true?

Declarative debugging

Computations are represented as trees (eg, proof trees) and debuggers search for "buggy nodes"

Eg, if a node is erroneous and all children are correct (an incorrect clause instance)

Or, if a node is erroneous with no erroneous children but at least one inadmissible child (a clause instance which results in a transition from admissible to inadmissible atoms)

Different definitions of inadmissible are possible, eg, the input arguments are ill-typed

Negation and missing answers

You can get away with saying inadmissible atoms are true in simple cases

But if

1. there are missing answers (not just wrong answers), or
2. there is negation in the program, or
3. you diagnose (and try to fix) more than one bug

diagnosis can be inaccurate

The solutions involve three truth values, explicitly or implicitly

Sixteen versions of even and odd

```
even(N) :- e4(N). % or e1/...
```

```
odd(N) :- o2(N). % or ...
```

```
e1(0).
e1(s(s(N))) :- e1(N).
```

```
o1(s(0)).
o1(s(s(N))) :- o1(N).
```

```
e2(0).
e2(s(N)) :- odd(N).
```

```
o2(s(N)) :- even(N).
```

```
e3(0).
e3(s(N)) :- not e3(N).
```

```
o3(s(N)) :- not o3(N).
```

```
e4(N) :- not odd(N).
```

```
o4(N) :- not even(N).
```

Programs have more than one meaning

An alternative description/intention for merge:

```
% merge(As, Bs, Cs): Cs is an interleaving of the lists
% of integers As and Bs and the number of runs in Cs is
% the maximum number of runs in As and Bs
```

An alternative set of possible modes:

```
% Either As and Bs or Cs should be input
```

Admissible if `As` and `Bs` or `Cs` are lists of integers

Three-valued semantics

The semantics attempts to reflect the programmer's intentions (which are imprecise)

Restricts programs as little as possible: for any given intention there are more "correct" programs

We use three truth values: *true*, $false$ and *inadmissible*

True atoms are intended to succeed, false atoms are intended to finitely fail

The user *doesn't care* about the behaviour of inadmissible atoms

Procedural semantics

Basically the same as SLDNF resolution (negated calls ground when selected)

A single "disjunctive clause" is used for each predicate definition (syntactic sugar, like Clark's completion):

$$
\begin{aligned}
merge(X, Y, Z) \leftarrow\ & \exists A \exists As \exists B \exists Bs \exists Cs \exists Bs \exists Cs \\
& (\quad X = [] \wedge Y = Bs \wedge Z = Bs \\
& \vee\ X = A.As \wedge Y = [] \wedge Z = A.As \\
& \vee\ X = A.As \wedge Y = B.Bs \wedge Z = A.Cs \wedge \\
& \qquad\qquad A \leq B \wedge merge(As, B.Bs, Cs) \\
& \vee\ X = A.As \wedge Y = B.Bs \wedge Z = B.Cs \wedge \\
& \qquad\qquad A > B \wedge merge(A.As, Bs, Cs) \\
&)
\end{aligned}
$$

Procedural semantics (cont.)

A variant of SLDNF trees is defined to better capture the search in Prolog/Mercury/...

A distinction is made between search for some and all solutions

```
p(X) :- not q(X).


q(a).
q(X) :- q(X).
```

A goal such as `p(a)` can have a finite search space (and we can use induction on the tree height to prove it has certain properties)

Fixed point theory

T_P was generalised to three-valued interpretations by Fitting as follows:

$T3_P(M)$ is the interpretation such that an atom A is

1. true, if there is a clause instance $A \leftarrow B$ where B is true in M,
2. false, if for all clause instances $A \leftarrow B$, B is false in M and
3. inadmissible, otherwise.

$T3_P$ has all the fixed points of T_P

Work of Fitting/Kunen relates this operator to procedural and model theoretic semantics

Fixed point theory (cont.)

We define operators which don't change the set of atoms we care about

$T3_P^+(M)$ treats inadmissible atoms as if they succeed:

$T3_P^+(M)$ is the interpretation such that an atom A is

1. inadmissible, if A is inadmissible in M,
2. true, if A is admissible and there is a clause instance $A \leftarrow B$ where B is true or inadmissible in M,
3. false, otherwise.

We previously related $T3_P^+$ to semantics for programs without negation

Fixed point theory (cont.)

$T3^-_P(M)$ is similar but treats inadmissible atoms as if they fail:

$T3^-_P(M)$ is the interpretation such that an atom A is

1. inadmissible, if A is inadmissible in M,
2. true, if A is admissible and there is a clause instance $A \leftarrow B$ where B is true in M,
3. false, otherwise.

$T3^+_P$ and $T3^-_P$ have all the fixed points of $T3_P$ and T_P

Model theory

Conjunction, disjunction and negation are defined as in Kleene's strong three-valued logic

$\wedge$	**T**	**F**	**I**
T	**T**	**F**	**I**
F	**F**	**F**	**F**
I	**I**	**F**	**I**

$\vee$	**T**	**F**	**I**
T	**T**	**T**	**T**
F	**T**	**F**	**I**
I	**T**	**I**	**I**

$\neg\mathbf{T} = \mathbf{F}$, $\neg\mathbf{F} = \mathbf{T}$, $\neg\mathbf{I} = \mathbf{I}$, $\exists$ is like $\vee$

The Fitting/Kunen semantics treats $\leftarrow$ in definitions as $\leftrightarrow$ (we call these *strong* models)

Our intended interpretations are *not* strong models of merge or 15 of the versions of even and odd

Model theory (cont.)

We use a weaker definition of a model based on insights from declarative debugging:

T $\leftarrow$ **T**	ok
F $\leftarrow$ **F**	ok
I $\leftarrow$ **I**	ok
F $\leftarrow$ **T**	wrong answers
F $\leftarrow$ **I**	(?) wrong answers
T $\leftarrow$ **F**	missing answers
T $\leftarrow$ **I**	(?) missing answers
I $\leftarrow$ **T**	ok!
I $\leftarrow$ **F**	ok!

$\leftarrow$	**T**	**F**	**I**
T	**T**	**F**	**F**
F	**F**	**T**	**F**
I	**T**	**T**	**T**

Three-valued semantics — Results

Interpretations can be ordered in various ways, including the information ordering:

$M_1 \subseteq_i M_2$ if $T_1 \subseteq T_2 \wedge F_1 \subseteq F_2$, where T_1 (F_1) and T_2 (F_2) are the true (false) atoms in M_1 and M_2

Propositions:

M is a model of $comp(P)$ iff $M \subseteq_i T3_P(M)$

M is a model of $comp(P)$ iff $T3_P^+(M) = M$ and $T3_P^-(M) = M$

If M is a model of $comp(P)$ all admissible instances of computed answers are $\mathbf{T}$ in M

If M is a model of $comp(P)$ and an all solutions computation of A terminates normally then every $\mathbf{T}$ instance of A in M is subsumed by some computed answer

Three-valued semantics (cont.)

Operational semantics	may succeed	must loop	may fail
$\subseteq_i$-least strong model	**T**	**I**	**F**
any strong model	**T**	**T/I/F**	**F**
any model	**T/I**	**T/I/F**	**I/F**

Checking M is the $\subseteq_i$-least strong model is hard

Checking M is a (strong) model is relatively easy — check each predicate definition separately

The difference in precision of strong models and models is only the behaviour of inadmissible atoms (which we don't care about)

The difference in flexibility is significant: there are natural models of merge and all versions of even and odd

Verification

```
subset1(L, M) :- not notsubset(L, M).

notsubset(L, M) :- member(X, L), not member(X, M).

member(X, [X|L]).
member(X, [Y|L]) :- member(X, L).
```

All but first argument of `member` are intended to be lists

Easy to check that (eg) if the head of the `subset1` clause is **T** (**F**) the body is **T** (**F**)

Verification (cont.)

Drabent and Miłkowska use two two-valued "specifications", one for soundness (atoms for which success is ok) and the other for completeness (atoms for which success is expected)

That is, the **T** and **I** atoms, and just the **T** atoms, respectively

The verification method uses original and primed versions of each predicate and priming and double priming operations on formulas

It very effectively obfuscates the truth tables of the three-valued logic

Verification (cont.)

```
subset2([], L).
subset2([H|T], LH) :-
    select(H, LH, L), subset2(T, L), not member(H, T).


select(H, [H|L], L).
select(H, [X|L], [X|LH]) :- select(H, L, LH).
```

Admissibility: second args of `subset2` and `select` are lists

First arg of `subset2` is a *duplicate-free* list whose elements are a subset of those in the second arg

We can show (eg) `subset2(X,[1,2,3])` doesn't miss any answers

Verification (cont.)

There is another model of `subset1` where it is inadmissible if there are duplicates in the first arg

This model has *less information* than the model of `subset2`, corresponding to the fact that `subset1` has less flexible modes

There are other models where both arguments are intended to be duplicate-free

Conclusion

How close are logic programs and specifications?

What is the weakest condition a programmer should enforce which ensures correct behaviour of their programs?

A1: A two-valued interpretation is a model of the program/completion

A2: A three-valued interpretation is a strong model

A3: A three-valued interpretation is a model