Exploring Founded Semantics for Static Analysis

Alex Gu

Introduction

In logic programming, one core issue is the presence of unrestricted negation. For example, in the simplest case, if we have a program `A :- ¬A`, there is no least fixed point solution to this program, and different semantics can disagree on the solution to this program. In this project, we explore one semantics designed to handle unrestricted negation known as *founded semantics* [1]. In short, founded semantics is a 3-valued semantics (True, False, Unknown) designed to provide one solution to the problem of unrestricted negation.

In this write-up, we first briefly introduce the idea behind founded semantics. Then, we show two toy examples highlighting the effectiveness of founded semantics on two static analyses, one concerning reachability, and one concerning strong updates. Finally, we conclude.

Founded Semantics

There are two main features of founded semantics: first, predicates can be declared as uncertain (T/F/U) or certain (T/F). Second, uncertain predicates can be defined as complete, which means we add in rules for its negation during inference.

Uncertain and Certain Predicates

A predicate is declared *certain* if each assertion has two values: T, F. For certain predicates, everything T must be given/inferred by following the rules, and the rest are F. A predicate is declared *uncertain* if each assertion has three values: T, F, U. Everything T/F must be given/inferred, and the rest are U.

For example, consider a program containing the statement `assign(line, variable, value)`, which represents an assignment of a variable to a value on a specified line. The predicate `assign` should be declared as certain, because we know that on a particular line, a variable either gets assigned to a value, or it doesn't. Now, consider a statement like `can_have_value(line, variable, value)`, a predicate that represents whether or not a variable can have a specified value on a given line. This might reasonably be declared as uncertain, because sometimes, we're not sure if a variable can have a given value on a particular line.

[1] Y. A. Liu and S. D. Stoller. Founded semantics and constraint semantics of logic rules. *arXiv preprint arXiv:1606.06269*, 2016.

Complete predicates
In this project, we declare all uncertain predicates as *complete*. For complete predicates, we add completion rules to define the negation of the predicate explicitly using the negation of all the hypotheses. For example, if A is declared complete, and we have a statement `A :- B ∨ C`, then we would add the rule `¬A :- ¬B ∧ ¬C`, corresponding to the negation of the original statement. Essentially, we use given predicates and these negated predicates to infer everything that is T or F, and leave everything else as U.

Computing founded semantics
The computation of founded semantics consists of several steps, which I will list below.

1. *Combine rules*: The first step, for each predicate, to combine all the rules defining that predicate. For example, if I had two rules `A :- B, A :- C`, I would combine them into a rule `A :- B ∨ C.` We also add in explicit instances of ∀ and ∃ quantifiers when necessary.
2. *Adding in negated predicates:* As mentioned earlier, the next step is to add in negation rules for uncertain predicates, and creating a new variable `n.A`, representing the negation of a variable A.
3. *Constructing the SCC graph*: Next, we construct a graph such that each predicate corresponds to one node. When we have a rule A depending on B, we add an edge from B to A. We then compute the strongly connected components of the graph, say S_1, S_2, …, S_n.
4. In order S_1, S_2, .., S_n, we compute the least fixed point of the predicates in each SCC. We essentially employ a worklist algorithm where for each node in the SCC, we infer all predicates we can from that node, and continue doing that until we've reached a fixed point.

Example 1: Reachability

One application where non-stratified negation causes issues is with demand-driven analyses. In demand-driven analysis, you might have two different analyses A and B, but only want to run A on some region of code if B declares that region of code as safe (for example, if it's guaranteed there are no division by 0 errors). We show a very simplified program that captures that idea.

In this example, we have an unsafe region of code modeled by `method_has_42(M)`, representing whether or not a method M might contain the constant 42. We'll model unsafeness as if a method doesn't contain 42. Here, the domain for M will just be `{“main”}`

```
output reach(string)
reach(“error”) :-
  reach(M),
  !method_has_42(M).
```

We'll say that a method has 42 if we can reach it, and the constant 42 is actually in the method:

```
output method_has_42(string)
method_has_42(M) :-
  reach(M),
  constant_in_method(M, 42).
```

Founded Semantics for Example 1
We first remark that in this example, there is an issue with non-stratified negation in this example, because `reach` depends negatively on `method_has_42`, which depends on `reach.` Let's see the result of founded semantics on this program. We'll consider `constant_in_method` as a certain predicate, and `reach, method_has_42` as an uncertain predicate.

Step 1: Combining

```
method_has_42(M) :- reach(M) ∧ constant_in_method(M, 42)
reach(“error”) :- ∃ M | reach(M) ∧ !method_has_42(M)
```

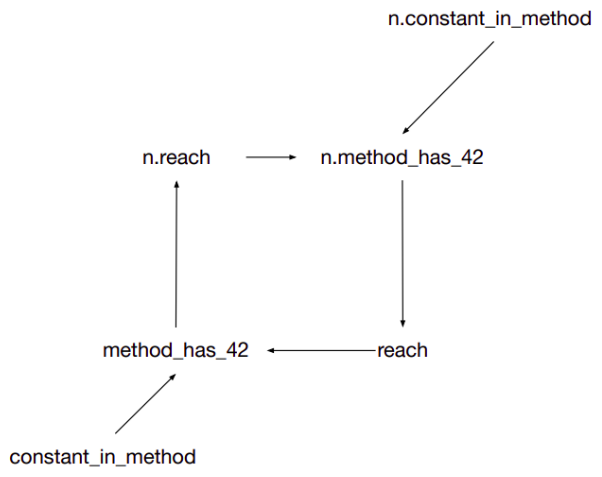
Step 2: Adding in negated rules

```
method_has_42(M) :- reach(M) ∧ constant_in_method(M, 42)
reach(“error”) :- ∃ M | reach(M) ∧ !method_has_42(M)
!method_has_42(M) :- !reach(M) ∨ !constant_in_method(M, 42)
!reach(“error”) :- ∀ M | !reach(M) ∨ method_has_42(M)
```

After renaming:

```
method_has_42(M) :- reach(M) ∧ constant_in_method(M, 42)
reach("error") :- ∃ M | reach(M) ∧ n.method_has_42(M)
n.method_has_42(M) :- n.reach(M) ∨ n.constant_in_method(M, 42)
n.reach("error") :- ∀ M | n.reach(M) ∨ method_has_42(M)
```

Step 3: Constructing the SCC
The SCC graph for the example looks as follows:


$$\begin{tikzcd}
 & & \text{n.constant\_in\_method} \arrow[dl] \\
\text{n.reach} \arrow[r] & \text{n.method\_has\_42} \arrow[d] & \\
\text{method\_has\_42} \arrow[u] & \text{reach} \arrow[l] & \\
\text{constant\_in\_method} \arrow[u] & &
\end{tikzcd}$$

Step 4: Computing the Least Fixed Point
To demonstrate the least fixed point computation, we'll add in two other rules

```
reach("main").
constant_in_method("main", 24).
```

First, we assume that 24 is the only constant in main, so it would be reasonable to assume `n.constant_in_method("main", 42)` is True and `!constant_in_method("main", 42)` is False.

The fixed point computation looks like this:

- First, since `n.constant_in_method("main", 42)` is true, we can then infer that `n.method_has_42("main")` is true
- We can then infer that `reach("error")` is true because `reach("main") ∧ n.method_has_42("main")`
- Now, we can infer that `method_has_42("main")` is false
- Finally, we infer `n.reach("error")` is false

You can see that this is a fixed point, and so everything else is declared as unknown. Therefore, we have inferred that `method_has_42("main") = F`, `reach(“error”) = T`, which is what we would expect (since “main” does not contain 42, and therefore, we would reach an error).

Example 2: Strong Update

Another scenario where you might run into issues with non-stratified negation is in a flow-sensitive points-to analysis with strong updates. Consider the following program:

```
A x = new A(); // heap allocation A
x.f = new B(); // heap allocation B
x.f = new C(); // heap allocation C
```

In this case, we know that x can only point to A, so by strong update, at the end of this program, we can assume that `A.f` points to C (rather than B). Now, consider another scenario where this may not be the case:

```
A x = new A(); // heap allocation A
x.f = new B(); // heap allocation B
if (*) { x = new D(); }
x.f = new C(); // heap allocation C
```

In this scenario, we no longer know that x has to point to A before the last line is executed, so it may be the case that `A.f` points to B. Therefore, strong update captures the sense that a variable might point to a single thing vs multiple things.

In this example, we’ll introduce a very simple example that captures the essence of strong update. Programs in this example consist of assignment statements to a single implicit variable. The analysis tries to figure out what values a variable might have at a given program point. To mimic strong updates, we add in an extra rule that if a variable has the value 42 at some program point, then the subsequent assignment may be ignored, leaving the variable with its previous value.

We have two types of input facts: `assignment(line, val)`, representing an assignment to a value `val` on a given line `line`, and `successor(line, line)`, representing that one line follows the other in the program.

The program is as follows:

```
type line = string
type val = i32

input assignment(line, val)
input successor(line, line)

output may_have_value(line, val)
output strong_update(line)

(* base case *)
may_have_value(Line, V) :-
  assignment(Line, V).

(* lack of strong update (flow-through) *)
may_have_value(Line, V) :-
  successor(Prev, Line),
  may_have_value(Prev, V),
  !strong_update(Line).

strong_update(Line) :-
  successor(Prev, Line),
  !may_have_value(Prev, 42).
```

Founded Semantics for Example 2:

Step 1: Combining

```
may_have_value(Line, V) :-
     assignment(Line, V) ∨ (∃ Prev | successor(Prev, Line) ∧
     may_have_value(Prev, V) ∧ !strong_update(Line))

strong_update(Line) :-
     ∃ Prev | successor(Prev, Line) ∧ !may_have_value(Prev, 42)
```

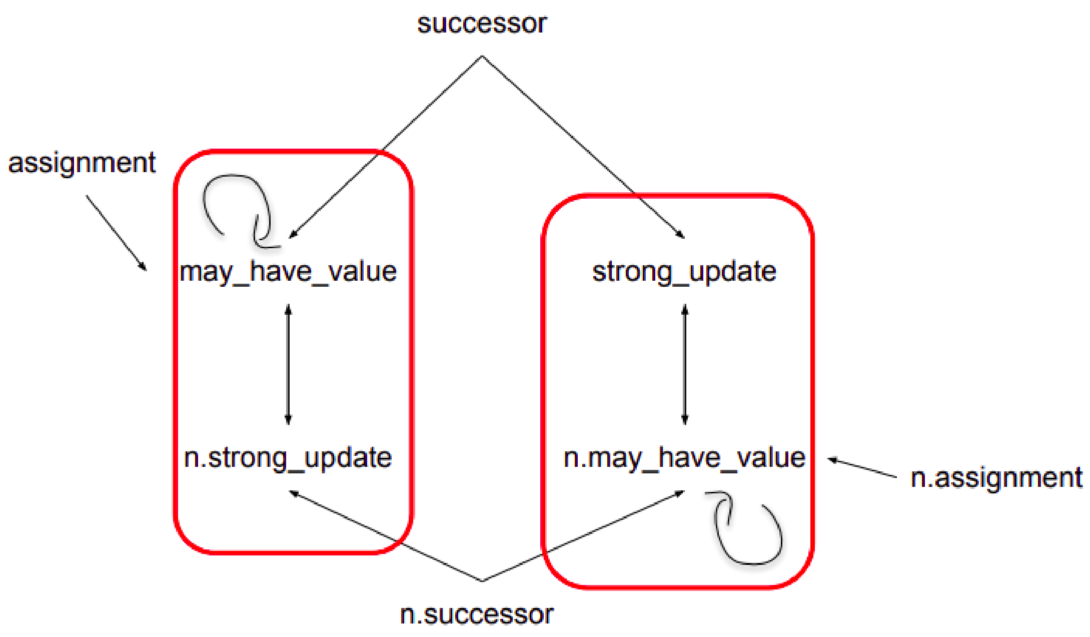
Step 2: Adding in negated rules and renaming

```
may_have_value(Line, V) :-
```

```
     assignment(Line, V) ∨ (∃ Prev | successor(Prev, Line) ∧
     may_have_value(Prev, V) ∧ n.strong_update(Line))

strong_update(Line) :-
     ∃ Prev | successor(Prev, Line) ∧ n.may_have_value(Prev,
42)

n.may_have_value(Line, V) :-
     n.assignment(Line, V) ∧ (∀ Prev | n.successor(Prev, Line)
     ∨ n.may_have_value(Prev, V) ∨ strong_update(Line))

n.strong_update(Line) :-
     ∀ Prev | n.successor(Prev, Line) ∨ may_have_value(Prev,
42)
```

Step 3: Constructing the SCC

The SCC for this example looks as follows:



Step 4: Computing the Least Fixed Point

We’ll compute the least fixed point for two sample programs. For this program, I implemented a short program in Python to do this, the code can be found here[1].

Program 1
```
assignment("A1", 0).
assignment("A2", 1).[1]
successor("A1", "A2").
```

For Program 1, we expect to see strong update at A2, and therefore A1 must be 0 and A2 must be 1. We expect to see these facts:
```
may_have_value("A1", 0).
may_have_value("A2", 1).
strong_update("A2").
```
After computing the least fixed point for this example, we get exactly this.

Program 2
```
assignment("B0", 42).
assignment("B1", 0).
assignment("B2", 1).
successor("B0", "B1").
successor("B1", "B2").
```

For Program 2, note that we don't have strong update for B1 or B2, because B0 may be 42. Therefore, B1 could be either 0 or 42, and B2 could be either 0, 1, or 42. Therefore, we expect to see these facts derived:
```
may_have_value("B0", 42).
may_have_value("B1", 42).
may_have_value("B1", 0).
may_have_value("B2", 42).
may_have_value("B2", 0).
may_have_value("B2", 1).
```

Again, after computing the least fixed point for this example, we get exactly this.

Conclusion

In both of these examples, we saw that founded semantics resulted in an analysis that matches our intuition. One surprising note about the second example is that rather than all four main predicates clauses being in one SCC, they were split up into two separate SCC's, for which the least fixed points could be computed independently. This project

[1] https://github.com/minimario/founded-semantics/blob/main/founded-semantics.ipynb

highlights the promising nature of founded semantics and its applications of static analysis.