The graph stratification version of the Marcel theorem prover: documentation and examples

Lavinia Randall Holmes

April 29, 2026

1 Introduction

This document introduces another version of the Marcel theorem prover for stratified set theory, so-called because it implements the system of sequent calculus for SF (stratified set theory with no extensionality axiom but with a set abstraction operator) defined by Marcel Crabbé and shown by him to satisfy cut elimination (the right rule for equality that we use actually gives the full extensionality of NF, and cut elimination is not known for this system, but this can be corrected if desired).

This particular version was immediately inspired by contemplation of the proposal by Ryan Nolan that the Bellman-Ford algorithm for finding shortest paths in weighted graphs in which edges might have negative weight could be used for stratification testing. Our actual proposal owes nothing in its details to Nolan because the use of that algorithm, while it works in principle, is a bad idea. We present a rather different algorithm to be applied to the graph on variables in the directed graph determined by the atomic formulas in a graph with weights determined by relative type and order of the variables, similar to Dijkstra's algorithm. The use of Bellman-Ford is a bad idea because the execution of this algorithm can be aborted the second time that any edge is "relaxed" (or if an overestimate of the distance is seen to exist, something that Bellman-Ford doesnt even notice) with a report of stratification failure.

It is also motivated by the concurrent and quite independent study by Thomas Forster and myself of the properties of the graph on variables in a formula determined by the atomic formulas in the graph (and in particular

the properties of the notion of connectedness for formulas determined by connectedness of this graph.

This version also follows the path of assigning invisible occurrence data to every variable in a term or formula. Occurrences of variables are typed, not the variables themselves, and this enables direct implemention of weak stratification (in fact, even weaker than in Marcel's system: all that is required is that variables connected to the binding variable in each set abstract have consistent types in relation to the binding variable.) This has (at least at this initial stage) the perhaps perverse feature that the definition of substitution can ignore variable capture: the reason is that the only instance in which occurrences of variables are regarded as identical is when they are bound by the same quantifier or set abstractor [sort of: clarification just below]. When substitutions are carried out, the substitution term has all of its occurrence information displaced to a new range (so that no identification can exist between any variable in the substituted term and any binder into whose scope it is substituted). The clarification is that identifications of occurrences *do* exist between variables in disjoint occurrences of the same substituted text, but this should not create difficulties. [it is possible for such identifications to cause displayed terms not to be stratified in a technical sense, but the prover only checks stratification at parse time, and such terms are always stratifiable by increasing diversity of variable occurrences]. Attention to the moral law probably requires us to install a warning to the viewer when free variables are introduced into scopes of binders using variables of the same shape: the distinction is not apparent to the viewer in term or formula display. But the prover knows and does not make errors due to apparent variable capture.

2 The language of the prover

For rapid prototyping, we wrote a parser using Polish notation, but the display function will look more familiar.

Terms are variables (letters `a-z` with one or more primes `‘` affixed) or set abstracts (discussed subsequently). The primes are prefix in the entry notation (to support the Polish nature of the parser) and postfix in the display notation. Variables have invisible occurrence indices: a new index is generated for each new variable, but then each occurrence of a variable bound by a quantifier or set abstractor is coerced to have the same occurrence index as the binding variable. Occurrence indices are also manipulated by

substitution and by the process of generating fresh variables in ways to be discussed below.

A set abstract in the input notation takes the form {`xf`, { being the abstraction operator, `x` being a variable, and `f` being a formula. In the display notation it takes the more familiar form $\{x \mid \phi\}$, ϕ being the same formula but in display notation.

Formulas are of various flavors we now list.

Atomic formulas are of the forms `=tu` (in display form $T = U$) and `etu` (display form $T\, e\, U$ in the prover: we might write $T \in U$) where `t` and `u` are terms with display forms T and U respectively. These are atomic assertions of equality and membership.

Boolean formulas are of the forms `Cfg` where C is a connective, one of `&,V,>,X` in the input language. Formulas `&fg, Vfg, >fg, Xfg` take the more familiar forms $(\phi \,\&\, \psi)$, $(\phi\, V\, \psi)$, $(\phi -> \psi)$ and $(\phi == \psi)$ respectively, where `f` and `g` are formulas with display forms ϕ and ψ. We may choose to write $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ for these familiar logical operations.

In a separate category due to its arity is the negation ∼`f` which is written $\sim \phi$ in display, where `f` is a formula with display form ϕ. We may also write $\neg\phi$.

Quantified formulas are of the forms `Axf`, `Exf` which are written $(Ax : \phi)$ and $(Ex : \phi)$ in display (where `f` is a formula with display forms ϕ)). We may also write the usual $(\forall x : \phi)$ and $(\exists x : \phi)$ here.

The `testt` and `testf` formulas can be used to practice term and formula entry.

```
testt("{x=xx")
'{x | x = x}'
testf("&exyeyz")
'(x e y & y e z)'
```

The user should be warned that the parser does stratification checks on set abstracts and will not allow set abstracts which are not weakly stratified in a suitable sense.

```
testt("{xexx")
x e x
['stratification failure', [['x', 14]], [['x', 14]], [['x', 14],
['x', 14], -1]]
'!!!'
```

Details to be explained later. But notice that the attempt to present $\{x \mid x \in x\}$ is a failure: one never even sees it, though one sees its body.

3 Substitutions

There are two kinds of substitution. One replaces all free instances of a particular occurrence of a variable in a context – this is used for operations on binding contexts (quantifier rules and comprehension). The other replaces all free variables with the same surface form (ignoring the occurrence index) in a context.

In both kinds of substitution, the term substituted in has all of its occurrence indices displaced above all existing text, so variables actually cannot be captured when substituted into binding scopes...but the display doesn't show this [something we ought to fix]. Later (after introducing the sequent prover) I'll put in an example of the weirdness of apparent variable capture.

4 The stratification algorithm

The first step of the stratification checker is to build a directed graph from the formula to be checked. Each atomic formula generates edges in both directions, with weight equal to the difference in relative type. $x = y$ creates edges with weight 0 from x to y and from y to x. $x \in y$ creates an edge from x to y with weight 1 and an edge from y to x with weight -1. Edges are also created by atomic formulas involving set abstracts. For example, $\{x \mid \phi\} \in z$ creates an edge from x to z of weight 2 and an edge from z to x of length -2, as well as an entire graph from ϕ.

Notice that distinct occurrences of free variables are distinct nodes in the graph. $x \in x$ is a (weakly) stratified formula, with the two occurrences of x having different relative types. But occurrences of bound variables with the same surface form and the same binder are identified: $\{x \mid x \in x\}$ raises a stratification error.

The stratification algorithm takes as input a node in the weighted graph and the entire graph. It computes distances from the selected node to all nodes connected with it. It does this by starting with a very rough estimate of the distance function (distance 0 from the selected node to itself [and the trivial path from the selected node to itself as an extra bit of data] and

infinite distance to all other nodes) and proceeds through the nodes of the graph improving the estimate at each step. When a node is encountered, if there is no finite distance from the selected node already computed, we move it to the end of the list and proceed to the next node. If there is a finite distance from the selected node to the current node, we compute new estimates of the distance from the selected node to each neighbor of the current node. If the distance to a neighbor is infinite, we get a finite estimate, and add that information and a path from the selected node to the neighbor [both computed in the obvious way] to the data for that neighboring node. If the distance to a neighbor is finite, and our new estimate of the distance is the same, we do nothing to that node. If the distance to a neighbor is finite and our new estimate is different from the original estimate (either an overestimate or an underestimate), we can stop the entire process and report stratification failure, with supporting data consisting of the path to the current node, the path to the neighboring node previously computed, and the edge between them. [This is why using the Bellman-Ford algorithm does not make sense: it will work, but the very first time that a distance is adjusted from a finite value to a smaller finite value, all subsequent work is a waste: and the Bellman-Ford algorithm entirely ignores overestimates, which are also evidence of failure.] When we arrive at the end of the list of vertices, we will have computed all finite distances from the selected node to other nodes, if no stratification error has been found, and we can return the partial distance function, which is in effect a relative type assignment witnessing stratification. Nodes not connected to the selected node by a path are not assigned distances.

The system actually uses the stratification algorithm only on set abstracts, ensuring that types can be assigned to each bound variable connected to the binding variable by a path in the graph in a consistent way. This is even weaker than the usual "weak stratification": for example
$\{x \mid x = x \wedge (\exists y : y \notin y)\}$ is stratified for our purposes.

```
testt("{x&=xxAy~eyy")
'{x | (x = x & (Ay : ~y e y))}'
```

The function `strattest` can be used to check a formula (not a term) for stratification.

```
testf("e{x=xx{x=xx")
'{x | x = x} e {x | x = x}'

strattest("e{x=xx{x=xx")
[['x', 27], [1], ['x', 23], 1, ['x', 27]]], [['x', 23], [0],
[['x', 23]]]]
```

In this term, there are two different occurrences of x, and the stratification data supplied gives distances for each of them and a path from the selected node (chosen automatically by the testing function) to each of the two nodes.

```
testf("eu{x&exy&eyzezx")
'u e {x | (x e y & (y e z & z e x))}'
strattest ("eu{x&exy&eyzezx")
[['z', 52], [-1], [['u', 46], 0, ['x', 47], -1, ['z', 52]]],
[['y', 49], [1], [['u', 46], 0, ['x', 47], 1, ['y', 49]]],
[['x', 47], [0], [['u', 46], 0, ['x', 47]]], [['u', 46], [0],
[['u', 46]]], [['y', 50], [], []], [['z', 51], [], []]]

testf("eu{xEyEz&exy&eyzezx")
(Ey : (Ez : (x e y & (y e z & z e x))))
['stratification failure', [['x', 55], 1, ['y', 56], 1, ['z',
57]], [['x', 55]], [['z', 57], ['x', 55], 1]]
'???'
```

Here we have two related examples. Neither are stratified in the strict sense, but the first is weakly stratified, because y and z are free, so different occurrences can be assigned different types.

In the second formula, where we quantify over y and z, there is no need to run the stratification checker: just trying to display the term with the unstratified set abstract in it is a sufficient disaster to cause the checker to complain.

To assist with reading the output: a successful stratification is a list of triples, a node followed by the singleton list of the distance to the selected

node (or the empty list to signify infinite distance) followed by a path (decorated with weights of edges) from the selected node to the given node. A report of failure contains two paths from the selected node and an edge between the termini of those two paths, witnessing a change in estimated finite distance.

The emphasis of this work has not been on creating nifty displays for stratification information: I could improve this, but have not done so so far.

5 The sequent prover

The remainder of the work is development of a user driven sequent prover for NF (which could be adapted to a prover for SF or NFU by changing the right equality rule). This parallels my earlier work on versions of the Marcel prover, but it does have interesting features.

The general plan of the sequent prover is that one uses `start s` to introduce a sequent of the form

$$\frac{}{P}$$

where P is the result of parsing the string s.
One then has a limited number of commands as options.

- The `right()` command applies an appropriate right rule to the first proposition on the right (below the line that is)
- The `left()` command applies an appropriate left rule to the first proposition on the left (above the line, that is).
- The `getleft(n)` command brings the nth proposition on the left (above the horizontal bar, actually) to the front.
- The `getright()` command brings the n proposition on the right to the front.
- The `done()` command invites the prover to recognize that the current sequent is an axiom (that the propositions at the beginnings of the lists on left and right are the same). Identity of propositions up to α-equivalence, ignoring occurrence data, is supported.

- The `setunknown(u,t)` command allows one to set the value (globally in the proof) of an "unknown" `u` (a free variable introduced by the right rule for the existential quantifier or the left rule for the universal quantifier) with a term `t`. In a usual presentation, a specific term is supplied when the rule is applied: here the unity of the `left()` and `right` rules is preserved by separating out the choice of witness. It may also be useful to delay choosing the witness until it is evident from developments in the proof what might work. There are limitations on what can replace an "unknown" [the term replacing it cannot contain any fresh variable introduced by quantifier rules which is introduced after the unknown which is being replaced].

I should add cut and some other utilities, but I have not as yet.

The prover maintains a stack of sequents needing attention. When the `left()` or `right()` rule is applied, one or two sequents are appended to the end of the proof [which is a list of sequents with points from sequents to ones justifying them] and one then proceeds to the first sequent on the stack.

When every line has a justification, the proof is complete.

The meat of the logic supported is in the functions `leftaction` and `rightaction`. which are used to indicate how to construct the one or two sequents from which a current sequent is derived.

We present the rules. I have copied these with some editing from a manual for a long-ago version of Marcel.

5.1 Specific Sequent Rules: Connectives

We give left rules and right rules (premise and conclusion rules) for the commonly used connectives, excluding converse implication and xor. The notations Γ and Δ represent arbitrary finite sets of propositions. The sentence closest to the turnstile on the right or left is the first sentence in the right or left list in the prover's presentation. The premise on the left is the one which is presented first by the prover when the rule is applied.

The left rule for the biconditional is lazily handled by definitional expansion.

In our experience the rule which it is somewhat difficult to get used to is the left rule for implication, though with a little thought it can be seen to express the rule of *modus ponens*.

5.2 Axiom

$$\Gamma, A \vdash A, \Delta$$

5.3 Left rules (Premise rules)

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}$$

$$\frac{\Gamma, A \to B, B \to A \vdash \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}$$

5.4 Right rules (Conclusion rules)

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}$$

$$\frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, B \vdash A, \Delta}{\Gamma \vdash A \leftrightarrow B, \Delta}$$

6 More Sequent Rules

For conventions on how rules are to be read in general, see the section on sequent rules for connectives above.

In the rules which follow, if ϕ is a formula, $\phi[t/x]$ is taken to represent the result of substituting the term t for the variable x in ϕ.

6.1 Rules for Quantifiers

6.1.1 Left Rules (Premise Rules)

$$\frac{\Gamma, \phi[t/x], (\forall x.\phi) \vdash \Delta}{\Gamma, (\forall x.\phi) \vdash \Delta}$$

where t is any term

$$\frac{\Gamma, \phi[a/x] \vdash \Delta}{\Gamma, (\exists x.\phi) \vdash \Delta}$$

where a is a variable not appearing in the conclusion

6.1.2 Right Rules (Conclusion Rules)

$$\frac{\Gamma \vdash \phi[a/x], \Delta}{\Gamma \vdash (\forall x.\phi), \Delta}$$

where a is a variable not appearing in the conclusion

$$\frac{\Gamma \vdash \phi[t/x], (\exists x.\phi), \Delta}{\Gamma \vdash (\exists x.\phi), \Delta}$$

where t is any term

6.1.3 Comments on Quantifier Rules

In the quantifier rules, we have retained the quantified sentence from the conclusion in the premise. This is so that we can avoid formalizing notions of copying and reordering formulas in sequents: a quantified formula may be reused several times in a proof, and if it were erased by the application of the rule we would need to copy it explicitly. Another advantage is that it preserves precise equivalence of the conclusion with the conjunction of all the premises, which is a feature of all the sequent rules of Marcel.

The rules requiring input of a new variable a supply a computer-generated variable. In the original version of this prover, the rules involving a new term t were implemented by separate commands with the term t as a parameter. In the current version, the computer supplies a new "unknown variable" which can subsequently be replaced by a term: the advantage is that the same command can then handle all basic sequent rules.

6.2 Rules for Membership

6.2.1 Left Rule

$$\frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, t \in \{x \mid \phi\} \vdash \Delta}$$

when ϕ is stratified (this condition is handled at parse time)

6.2.2 Right Rule

$$\frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash t \in \{x \mid \phi\}, \Delta}$$

when ϕ is stratified (this condition is handled at parse time)

6.3 Rules for Equality

6.3.1 Left Rule

$$\frac{\Gamma, (\forall v : t \in v \leftrightarrow u \in v) \vdash \Delta}{\Gamma, t = u \vdash \Delta}$$

This is just definitional expansion. It allows substitution only in stratified contexts, but this is enough for the full strength of substitution as usually presented.

6.3.2 Right Rule

$$\frac{\Gamma \vdash (Ax.x \in t \leftrightarrow x \in u), \Delta}{\Gamma \vdash t = u, \Delta}$$

This gives full extensionality. It can be revised to give SF (use the same Leibniz formulation as in the left rule) or NFU, a bit more elaborate.

6.4 Global Substitution

The left rule for the universal quantifier and the right rule for the existential quantifier require input of a specific term t. What the prover actually supplies is an fresh variable identified as an "unknown", annotated with the list of all fresh variables generated before it.

The `setunknown` command allows the user to set the unknown variable to a particular term. There is a restriction on what terms can replace an unknown variable: the term must have been definable at the time that the unknown variable was generated, so it cannot contain any free or unknown variables which were not already declared at the time it was introduced (this data is stored with the unknown variable).

It is worth noting that the display function suffixes arbitrary variables with ! and unknown variables with ?, and it forbids variables of the same shape which are bound from being introduced. The parser does not require or even understand these suffixes.

An advantage of this approach are that it enables one to delay the choice of a witness: the later progress of the proof may make it more evident what witness will work.