Tableau-Based Model Generation for Relational Syllogsitic Logics

Erik Wennstrom

November 22, 2013

Abstract

I present an analytic tableau system for a small fragment of natural language called $\mathcal{RC}^\dagger$. $\mathcal{RC}^\dagger$ is an extension of the classical Aristotelian syllogisms ("All computer-scientists are mathematicians") to include relations in the form of transitive verbs ("All who hate all logic programmers know some who hate all proof theoreticians") and to permit the negation of nouns ("Some non-linguists love all set-theorists.") This logic is big enough to be interesting, but it's small enough that we can actually get some results. In this case, the results are that my tableau system (with an appropriate growth strategy) can be used to generate finite models for any satisfiable set of formulas from $\mathcal{RC}^\dagger$ in a finite amount of time. And if a set is *un*satisfiable, then the strategy will, in finite time, result in a closed tableau, proving its unsatisfiability. I also discuss $\mathcal{RC}$-tabModG, an implementation of the tableau system in MiniKanren (a relational programming system embedded in Scheme).

1 Introduction

This work falls into the are of what Larry Moss calls *natural logic*, a logical analysis of fragments of natural language that are big enough to be interesting, but small enough that we can get our hands on them. In the case of this paper, I am interested in *relational syllogistic logics*, which extend the traditional Aristotelian syllogistic logic $\mathcal{S}$ (the logic of sentences like "All computer-scientists are mathematicians." and "Some AI-researchers are logicians.") by introducing relational terms (allowing sentences like "All who hate all logic programmers know some who hate all proof theoreticians.") While we're at it, we'll also add the negation of nouns (which enables sentences like "Some non-linguists love all set-theorists.") We call the resulting logic $\mathcal{RC}^\dagger$. (The † symbol is read "dagger.")

Reinhard Muskens was the first to suggest using tableau systems for reasoning about natural logic.[11] This article has a somewhat different focus than Muskens', but there are many similarities, and I've adopted (and adjusted) some of his notation here. Muskens wanted his system to encompass a very wide range of reasoning about a large fragment of natural language, while I've taken Moss's approach of fixing a (relatively) small fragment and ensuring that my system is as complete as it can be.

In order to "get our hands" on these logics, I introduce a system of *analytic tableaux* designed to generate finite models for sets of $\mathcal{RC}^\dagger$ formulas or, when there aren't any, to provide inconsistency proofs. Tableaux have been used both to prove theorems (by proving the inconsistency of their negations) and to generate models, but the strategies used for each are not the same. The primary goal of the system I present here is to generate models, so I avoid some of the techniques that are usually used to speed up the search for inconsistency proofs (such as Skolemization and free-variable tableaux). However, the system is designed so that there are growth strategies that will always terminate in finite time, either with a model, or a closed tableau.

I also describe an implementation of the system in MiniKanren, a relational programming system embedded in Scheme. My implementation makes use of MiniKanren to investigate each branch of a tableau more-or-less in parallel. MiniKanren is a relational language, so my implementation can also be used to verify the correctness of tableaux or their branches.

In section 2, I'll provide the necessary preliminaries for the logics I'll be working with and for analytic tableau systems in general. In section 3, I'll lay out the details of my system and discuss various different growth strategies. In section, 4, I'll discuss the MiniKanren implementation and talk about its performance.

2 Preliminaries

2.1 Relational Syllogistic Logics

Let's get more specific about the logics we'll be dealing with here. The definitions and abbreviations here come primarily from Lawrence Moss and are mostly the same as you see in [13], although what they call $\mathcal{R}^*$ in that paper, I call $\mathcal{RC}$.[1] All of these logics begin with a set $\mathbf{P}$ of *unary atoms* or *nouns*.

In the traditional Aristotelian syllogistic logic, which we call $\mathcal{S}$, for "$\mathcal{S}$yllogisms," sentences are of the form All a are b, Some a are b, No a are b, or Not all a are b, where a and b are nouns in $\mathbf{P}$. Here, I'll introduce some space-saving notation (the space-savings are modest for $\mathcal{S}$, but the notation will extend very nicely to more complex logics, where it makes a bigger difference.) The sentence All a are b is abbreviated $\forall(a, b)$, and Some a are b is abbreviated $\exists(a, b)$. No a are b could be written $\nexists(a, b)$, but it will be convenient to push the negation down to the level of the nouns. So No a are b is abbreviated $\forall(a, \overline{b})$ and Not all a are b is abbreviated $\exists(a, \overline{b})$. In $\mathcal{S}$, negation of the subject noun (e.g., $\forall(\overline{a}, b)$) is not allowed.

$\mathcal{S}$ can be extended to $\mathcal{S}^\dagger$ by allowing the negation of nouns everywhere, so that sentences like All non-a are b (abbreviated $\forall(\overline{a}, b)$) or Some non-a are non-b

[1]I'm following the notation Moss currently uses.

(abbreviated $\exists(\overline{a}, \overline{b})$.[2] Note that since No a are b, is equivalent to All a are non-b ($\forall(a, \overline{b})$), the No quantifier does not need to be dealt with explicitly. The same goes for the Not all quantifier.

$\mathcal{S}$ can be extended to include verbs other than the copula *to be*. Specifically, the logic $\mathcal{R}$ (for "$\mathcal{R}$elational") adds transitive verbs to allow us to reason about binary relations. In addition to the set $\mathbf{P}$ of unary atoms, there is also a set $\mathbf{R}$ of *binary atoms*, otherwise known as *transitive verbs* (*tvs*). The new sentences are all of the form Q_1 $noun_1$ $verb$ Q_2 $noun_2$, where Q_1 and Q_2 are quantifiers (either all, some, no, or not all), $noun_1$ and $noun_2$ are nouns in $\mathbf{P}$, and $verb$ is a transitive verb in $\mathbf{R}$. This gives us sentences like All a *see* some b, Some a *see* no b, and Not all a *see* all b.

When it comes to the abbreviated notation, we push the negations on the quantifiers down to the verb-level. So the sentence Not all a *see* some b is first rewritten as Some a fail-to-*see* all b, and Some a *see* no b is rewritten as Some a fail-to-*see* all b. Unfortunately, I can think of no way to avoid the ambiguity present in sentences like these while still retaining the narrow-scope negation of the verb. The meaning intended by Some a fail-to-*see* all b would be better phrased Some a *see* no b, and All a fail-to-*see* some b is intended to mean the same thing as All a have some b that they don't *see*. Once we've dealt with any negated quantifiers, then the sentence Q_1 $noun_1$ $verb$ Q_2 $noun_2$ is written $\mathsf{Q}_1\big(noun_1, \mathsf{Q}_2(noun_2, verb)\big)$. The way to think about it is to think of something like $\forall(a, see)$ as a unary predicate describing things that *see* all a. The term $\exists(a, see)$ describes things that *see* some a. So a sentence like All a *know* some b would be abbreviated $\forall\big(a, \exists(b, know)\big)$, and All a fail-to-*love* all b is abbreviated $\forall\big(a, \forall(b, \overline{love})\big)$. $\mathcal{R}$ is extended to $\mathcal{R}^\dagger$ in the same way that $\mathcal{S}$ was extended to $\mathcal{S}^\dagger$: by allowing the negation of noun terms in addition to the verbs.

If you've been paying attention to the notation, you may be able to guess the next extension I've got in mind. When we went from $\mathcal{S}$to $\mathcal{R}$, we allowed replacing the second noun with a more complex unary predicate, but we did not do the same for the first noun. There's nothing stopping us from creating a logic (call it $\mathcal{RC}$, for "$\mathcal{R}$elative $\mathcal{C}$lauses") where we allow such sentences. For example, consider the sentence $\forall\big(\exists(a, love), \forall(b, like)\big)$. But this is *natural* logic, after all, so we better make sure that such a sentence corresponds to something in natural language. And indeed, it matches up nicely with a specific kind of *relative clause*. The example I gave would be an abbreviation for the English sentence All who *love* some a *like* all b.[3] To be clear, nouns are allowed to be either unary atoms from $\mathbf{P}$ or relative clauses of the form who $verb$ Q $noun$,

[2]For noun-level negations in English, I'm going to use the prefix non- (instead of moving the negation to the verb) to try and avoid the ambiguity present in sentences like All a are not b. It's sometimes possible to lessen ambiguity by pushing the negations all the way out to the quantifier (e.g., No a are b,) but the noun-negations match the abbreviated notation more closely, so I'll usually stick with that.

[3]You may have noticed that for simple examples, while I often use single-letter variables for the unary atoms, I will usually use English verbs for the binary atoms. The reason is that I find it slightly harder to parse single-letter variables as transitive verbs in these contexts. So a sentence like Some who r all a s all who r some b is harder for me to think about than one like Some who *hate* all a *know* all who *hate* some b.

where *verb* is a transitive verb from $\mathbf{R}$ or its negation, Q is all or some, and *noun* is another noun.[4] Note that *noun* might be a unary atom, or it might be a relative clause itself. This means that we can have clauses of arbitrary depth, including sentences like All who *fear* all who *see* all who *hate* some *intuitionists fear* all who *see* all who *hate* some *logicians*, which can only really be considered *natural* in the most theoretic of ways. The sentence's abbreviation $\forall\Big(\forall\big(\forall(\exists(ints, hate), see), fear\big), \forall\big(\forall(\exists(logs, hate), see), fear\big)\Big)$ is shorter to write (and easier to manipulate, as you'll see when I define the tableau system), but it doesn't make it any easier to think about. As before, we can allow the negation of nouns to get the logic $\mathcal{RC}^\dagger$, which is the primary logic I'll be working with in this paper.

The semantics for $\mathcal{RC}^\dagger$ (and its sublogics) is fairly straightforward. A model $\mathcal{M} = \langle M, [\![\cdot]\!]\rangle$ consists of a set M and a map $[\![\cdot]\!]$ that sends the unary atoms of $\mathbf{P}$ to subsets of M and the binary atoms of $\mathbf{R}$ to binary relations over M. We can easily extend $[\![\cdot]\!]$ to model any unary or binary predicate of $\mathcal{RC}^\dagger$. For any unary predicate p, $[\![\overline{p}]\!] := M - [\![p]\!]$, and for any binary predicate r, $[\![\overline{r}]\!] := M^2 - [\![r]\!]$. For unary predicate p and binary predicate r, $[\![\forall(p,r)]\!] := \{x \in M \mid \langle x,y\rangle \in [\![r]\!] \text{ for all } y \in [\![p]\!]\}$ and $[\![\exists(p,r)]\!] := \{x \in M \mid \langle x,y\rangle \in [\![r]\!] \text{ for some } y \in [\![p]\!]\}$. A model $\mathcal{M}$ *satisfies* a universal formula $\forall(p,q)$ (written $\mathcal{M} \models \forall(p,q)$) when $[\![p]\!] \subseteq [\![q]\!]$, and it satisfies an existential formula $\exists(p,q)$ ($\mathcal{M} \models \exists(p,q)$) when $[\![p]\!] \cap [\![q]\!] \neq \varnothing$. A model $\mathcal{M}$ satisfies a set Δ of formulas ($\mathcal{M} \models \Delta$) when it satisfies every member of Δ.

I've described these natural syllogistic logics as "small enough that we can get our hands on them," and now it's time for me to explain what I mean by that. First, let's talk about proof systems for the logics. Both $\mathcal{S}$ and $\mathcal{S}^\dagger$ have complete and *direct* syllogistic proof systems, meaning that proofs can be written using only sentences in the logic and without making use of any form of proof by contradiction. $\mathcal{R}$ and $\mathcal{RC}$ also have complete syllogistic proof systems, but those systems require the use of proof by contradiction. In the case of $\mathcal{R}$, you can prove any valid theorem using a single *reductio ad absurdum* step at the very end of the proof. The logics $\mathcal{R}^\dagger$ and $\mathcal{RC}^\dagger$ require non-syllogistic proof systems, ones that make use of variables in the intermediary steps. (All these results are from [13], as are most of the complexity results below.)

Now you might be asking why I'm not just translating everything into first-order logic (FO) and work with that. The obvious answer to that (in addition to the ability to write syllogistic proofs for some of these fragments) is that the decision problem for FO isn't even decidable. If you're clever, you might notice that all of these fragments embed into *two variable* first-order logic (FO_2), which is easier to work with, most notably because the decision complexity for determining whether a formula of FO_2 is a theorem is complete for non-deterministic exponential time (NExp).[7] From a complexity standpoint, these

[4]Note that we're allowing relative clauses to act as nouns, not to *modify* existing nouns. This allows noun phrases like All who *hate* some *logician* but *not* noun phrases like All *linguists* who *hate* some *logician*. In English, these two kinds of relative clauses happen to be written using the exact same words, which can make things a bit confusing.

fragments are all easier to work with, some of them much more so. $\mathcal{S}$, $\mathcal{S}^\dagger$, and $\mathcal{R}$ are all N-complete (non-deterministic logarithmic space). $\mathcal{RC}$ is coNP-complete [9] (the complement of nondeterministic polynomial time), and both $\mathcal{R}^\dagger$ and $\mathcal{RC}^\dagger$ are Exp-complete (exponential time).

But the fact that all these logics are fragments of $\textsc{fo}_2$ gets us another nice property, one which I will be taking heavy advantage of in this paper. Namely, $\textsc{fo}_2$ has the *finite model property*, meaning that any satisfiable formula of $\textsc{fo}_2$ is satisfied by a *finite* model.[10] Any set of sentences in $\mathcal{RC}^\dagger$ can be translated into a first-order logic formula using only two variables, so every set of $\mathcal{RC}^\dagger$ formulas is either inconsistent or is satisfied by a *finite* model. This fact is what makes the tableau system I'll later define such a good fit for $\mathcal{RC}^\dagger$ and fragments thereof.

2.2 Tableau Systems

When Moss first suggested that analytic tableaux[5] (also called *semantic tableaux*) for natural syllogistic logics might be worth investigating, I had never heard the phrase, and I thought they were some mysterious new proof theory technique. It turns out that I'd been teaching the method of analytic tableaux to freshman informatics students for years, only in undergraduate logic courses, they're called *truth trees*.

To give you a flavor, here's a simple example of an analytic tableau for the propositional formula $(A \lor B) \land \neg(A \to B)$.

$$\begin{tikzcd}[row sep=small]
& (A \lor B) \land \neg(A \to B) \arrow[d, dash] & \\
& A \lor B & \\
& \neg(A \to B) \arrow[dl, dash] \arrow[dr, dash] & \\
A \arrow[d, dash] & & B \arrow[d, dash] \\
A & & A \\
\neg B & & \neg B \\
\circ & & \times
\end{tikzcd}$$

The tableau is grown downward from the top according to rules for each binary connective. I won't lay out all the rules for propositional logic tableaux because that's not our focus. But the way I've always taught my students to think about this is as a formalized, methodical attempt to find a model that satisfies the formula. So if we want to satisfy the conjunction $(A \lor B) \land \neg(A \to B)$, we must satisfy both of its conjuncts, and so we stack those requirements up in a single branch. Next, to satisfy $A \lor B$, we have the option of trying to satisfy A or of trying to satisfy B, resulting in two separate branches. And finally, regardless of which branch we investigate, satisfying $\neg(A \to B)$ requires satisfying both A and $\neg B$.

Once every non-atomic formula in the tableau has been dealt with, it's a

[5]The plural of *tableau* is *tableaux*, pronounced the same as the singular. Blame the French language, if you must.

simple matter of investigating the atomic formulas in each branch and seeing if any of them contradict each other. In the right branch, it's not possible to satisfy both B and $\neg B$, so we close off the branch (thus the $\times$ at the end). In fact, we don't have to wait until we've finished to close off branches, and it will always be a good strategy to close off a branch as soon as a direct contradiction appears in it. The left branch of this example does not contain any contradictions, and we declare it to be open (indicated by $\circ$). Such a branch that can be expanded no further but hasn't been closed is said to be *saturated*.[6] A saturated branch represents a successful attempt at satisfying the formula $(A \vee B) \wedge \neg(A \rightarrow B)$. In this case, it will be satisfied by any model that assigns True to A and False to B. In this case, there's only one such model, but it's not uncommon to have open branches that do not fully specify a model, and in those cases, it doesn't matter what you do with the rest of the model, as long as it satisfies all the atomic formulas.

Due to the systematic nature of the process, a failure to find an open branch means that the root formula or formulas are unsatisfiable. This is what is meant by *soundness* for a tableau system. More precisely, a tableau system is said to be *sound* for a logic if whenever there is a closed tableau (i.e., one where every branch is closed) with root formulas Δ, then the set Δ is unsatisfiable. A tableau system is said to be *(weakly) complete* when we have things the other way around: if a set Δ is unsatisfiable, then there is a closed tableau for Δ. *Strong completeness* is much rarer and requires that *any* strategy for growing a tableau for an unsatisfiable set of formulas will result in a closed tableau. Propositional logic is one of the few logics that has a strongly complete tableau system. Different strategies (i.e., different ways of deciding which formula to expand when) can result in smaller or larger tableaux, but the end result is always the same.

For a logic like FO, however, things aren't so nice. Let's look at some of the growth rules for the standard tableau system for FO. There are other rules for the propositional connectives, but this should be enough to get a flavor for it. Note that the universal formulas (this includes negated existential formulas) can be expanded more than once. In order to saturate a branch and declare it open, one would have to expand every universal formula for every single variable that exists in that branch. A simple universal-existential formula like $\forall x \exists y . P(x, y)$ easily creates an infinite loop of universal instantiation followed by existential creation. If this is one of your root formulas and you're trying to close off the tableau, you might have to be careful about the order in which you choose to expand things. (This tableau system is complete, but not *strongly* complete.[8]) And if your root formulas are satisfiable, you'll never figure this out because you'll never reach a saturated branch.

If your goal is just to prove theorems (by showing their negations are unsatisfiable), there are things you can do to the tableau system to improve efficiency, such as Skolemizing away all the existential or switching to a free-variable

[6]Boolos calls such branches *finished* in [2], but I've adopted the terminology that Dellert uses in [4].

Figure 1: Tableau Rules for FO

$$\begin{array}{cccc}
\forall x.P(x) & \neg\,\forall x.P(x) & \exists x.P(x) & \neg\,\exists x.P(x) \\
| & | & | & | \\
P(t) & \neg\,P(c) & P(c) & \neg\,P(t) \\
\text{for existing term } t & \text{for new variable } c & \text{for new variable } c & \text{for existing term } t
\end{array}$$

tableau system. But these methods don't really help with model generation, so I won't be adopting them in this paper. I will try to create systems and strategies that terminate when there are no models, but they may not do so very quickly.

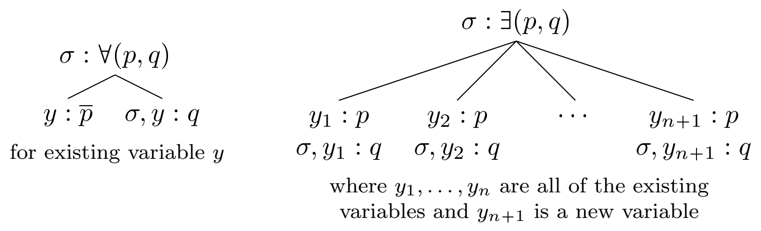
The tableau rules that I will introduce for $\mathcal{RC}^{\dagger}$ in section 3 are similar to what you'd get if you translated each $\mathcal{RC}^{\dagger}$ formulas into FO and then expanded the formula according to tableau rules for FO. But since my primary goal is model generation, the existential rule is modified to take advantage of the branching nature of tableaux to try and find finite models. The existential rule has several branches, each one investigating a different candidate for the existential variable in question. There are branches for each of the variables that already exist in the branch plus an extra branch considering the possibility that a new constant variable is required. While I believe that such a technique is novel in the context of natural logics, a very similar method for FO was proposed by John Burgess and proven to always find finite models for FO formulas when they exist by Boolos.[2]

3 A Tableau System for $\mathcal{RC}^{\dagger}$

The nodes in these tableaux consist of a list of variables followed by an $\mathcal{RC}^{\dagger}$ predicate of the appropriate arity. In the case of $\mathcal{RC}^{\dagger}$ formulas (which are nullary predicates), I will usually just omit the (empty) list. The rules for growing tableaux can be phrased in terms of natural language fragments, but adopting the abbreviated notation described in section 2.1 makes them much easier to write down. In these rules, σ is a (possibly empty) list of variables. (σ, x is meant to be read as a concatenation of the item x on to the list σ, so if σ is the empty list, σ, x is just x.) If we're dealing with $\mathcal{S}$ or $\mathcal{S}^{\dagger}$, σ will always be empty, and in the case of $\mathcal{R}$, $\mathcal{R}^{\dagger}$, $\mathcal{RC}$, and $\mathcal{RC}^{\dagger}$, σ will either be empty or unary. Two-element lists are possible, but only as literals (e.g., $x, y : r$ or $x, y : \overline{r}$). The overline notation for the negation of atoms is extended to more complex predicates in the natural way: $\overline{\forall(p, q)} := \exists(p, \overline{q})$ and $\overline{\exists(p, q)} := \forall(p, \overline{q})$.

In case the bare definitions are a little confusing, I'll share how I think about them. Suppose we're set to expand the formula $x : \forall(p, see)$ for the existing variable y. This formula essentially says that x *see*s all p. So either y is a non-p (represented by the left branch $y : \overline{p}$) or x *see*s y (represented by the right branch $x, y : see$). For the existential rule, suppose we're expanding the formula $x : \exists(p, see)$, which says that x *see*s some p. If $y_1, \ldots, y_n$ is the list of all the

Figure 2: Tableau Rules for $\mathcal{RC}^\dagger$



variables that exist in the branch so far, then either one of those y_i's serves as the p that x *see*s (resulting in the formulas $y_i : p$ and $x, y_i : \mathit{see}$) or none of them do, and it's some new y_{n+1} that is the p that x *see*s.

It will be useful to extend the semantics to deal with the variables used in the tableaux. An *extended* model $\mathcal{M}^* = \langle M, [\![\cdot]\!]\rangle$ is an ordinary $\mathcal{RC}^\dagger$ model with its denotation map $[\![\cdot]\!]$ extended to assign each variable to an element of M. Semantic entailment is also extended. For a model $\mathcal{M}^*$, a variable x and a unary predicate p, $\mathcal{M}^* \models x : p$ when $[\![x]\!] \in [\![p]\!]$, and or a model $\mathcal{M}^*$, variables x and y and a binary predicate r, $\mathcal{M}^* \models x, y : r$ when $\left\langle [\![x]\!], [\![y]\!] \right\rangle \in [\![r]\!]$.

Theorem 3.1 (Soundness)**.** *Given a set Δ of $\mathcal{RC}^\dagger$ formulas, if there is a closed tableau with root formulas Δ, then Δ is unsatisfiable.*

Proof. Suppose that we have a closed tableau with root formulas Δ, but Δ *is* satisfied by some model $\mathcal{M} = \langle M, [\![\cdot]\!]\rangle$. I will find a contradiction by showing that there is some branch of the tableau whose every node is satisfied by the model $\mathcal{M}^*$ (where $\mathcal{M}^*$ is an extension of the model $\mathcal{M}$). This is Lemma 3.2. Since the tableau is closed and therefore the branch contains contradictory nodes, this is impossible. $\square$

Lemma 3.2. *Given a set of $\mathcal{RC}^\dagger$ formulas Δ satisfied by some model $\mathcal{M} = \langle M, [\![\cdot]\!]\rangle$ and a tableau with with root formulas Δ, there is a branch of the tableau and an extension $\mathcal{M}^*$ of $\mathcal{M}$ such that $\mathcal{M}^x$ satisfies every node in the branch.*

Proof. We inductively extend the model $\mathcal{M}$ to $\mathcal{M}^*$, simultaneously determining which branch it will satisfy. By assumption, $\mathcal{M} \models \Delta$, so $\mathcal{M}$ satisfies all the root formulas. Suppose we have partially defined $\mathcal{M}^*$ up to a point, so that $[\![x]\!]$ is defined for every constant variable x that has appeared so far, and so that it satisfies every node of the branch so far. There are several (four, to be precise) cases, depending on whether the next rule applied is a universal or existential formula, and depending on how many variables are in the formula. I will prove two of the cases explicitly; the rest are very similar.

Suppose we expand the formula $c : \forall(p, r)$ for the variable d. By assumption $\mathcal{M}^* \models c : \forall(p, r)$, so we know that $[\![c]\!] \in [\![\forall(p, r)]\!]$, and so for all $y \in [\![p]\!]$, we know that $\left\langle [\![c]\!], y \right\rangle \in [\![r]\!]$. If $[\![d]\!] \notin [\![p]\!]$, then $\mathcal{M}^* \models d : \overline{p}$, and we move to the left branch. Otherwise, we must have that $\left\langle [\![c]\!], [\![d]\!] \right\rangle \in [\![r]\!]$, and so $\mathcal{M}^* \models c, d : r$ and we move to the right branch.

Now let's consider an existential case. Suppose we expand the formula $c : \exists(p, r)$ and we have existing variables $d_1, \ldots, d_n$. Since $\mathcal{M}^* \models c : \exists(p, r)$, so we know that $[\![c]\!] \in [\![\exists(p, r)]\!]$, i.e., there is some $y \in [\![p]\!]$ with $\left\langle [\![c]\!], y \right\rangle \in [\![r]\!]$. We might be able to pick one of the branches using old variables, but we don't need to. Instead we can always move along the rightmost branch, which introduces the new constant variable d_{n+1}. We define $[\![d]\!]_{n+1} := y$ and so we have that $\mathcal{M}^* \models d_{n+1} : p$ and $\mathcal{M}^* \models c, d_{n+1} : r$. $\square$

If the model $\mathcal{M}$ is a *finite* model, than we can actually guarantee that the extension $\mathcal{M}^*$ satisfies every node in a *finite* branch of the tableau. We simply follow the above strategy until there are as many existing variables $(d_1, \ldots, d_n)$ as the size of the model $|M|$. After that point, when we come to an existential rule, the pigeonhole principle ensures that we no longer have to choose the rightmost branch. One of the branches that reuses a constant variable is acceptable. Since the branch we are on no longer introduces new variables, there are only finitely many formulas and formula-variable pairs to expand and eventually, the branch will be saturated. This last property is what Boolos calls being *complete for finite satisfiability*, in contrast to the standard notion of completeness, which he calls being *complete for unsatisfiability.*[7]

Theorem 3.3. *Given a set of $\mathcal{RC}^\dagger$ formulas Δ satisfied by some finite model $\mathcal{M} = \langle M, [\![\cdot]\!] \rangle$ and a tableau with with root formulas Δ, there is a finite branch of the tableau and an extension $\mathcal{M}^*$ of $\mathcal{M}$ such that $\mathcal{M}^x$ satisfies every node in the branch.*

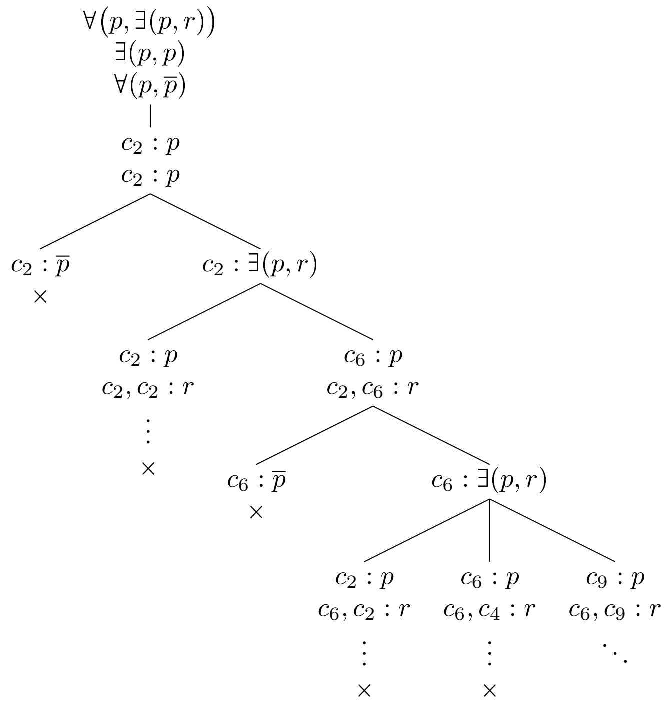
This is a *strong* result in that it doesn't matter in what order the formulas are chosen to grow the tableau. If the formulas are finitely satisfiable, any breadth-first search will uncover a finite model in a finite amount of time. Given that $\mathcal{RC}^\dagger$ has the finite model property, any breadth-first tableau strategy will prove a satisfiable set of formulas to be satisfiable in a finite amount of time. However, depending on the growth strategy, unsatisfiable formulas can result in infinite tableaux.

3.1 Completeness and the Fifo Strategy

The tableau system is strongly complete for $\mathcal{S}$ and $\mathcal{S}^\dagger$. Infinite branches are impossible because the only formulas introduced by growth rules in $\mathcal{S}^\dagger$ are literals (which can't be expanded any further). So once you've expanded all the existential root formulas, there are no new variables, and so there are only finitely many variables to be expanded for each of the universal formulas. Eventually, every branch will either be closed or saturated. But for the relational fragments like $\mathcal{RC}^\dagger$, or even $\mathcal{R}$, we aren't so lucky. Consider the set of formulas $\Delta = \left\{ \forall\big(p, \exists(p, r)\big), \exists(p, p), \forall(p, \overline{p}) \right\}$. Clearly this set is unsatisfiable due to the

[7]To me, the property seems to have more in common with soundness than completeness.

last two formulas, but a willfully obtuse growth strategy can result in an infinite tableau. (I've named the constant variable introduced by the nth formula in a branch c_n.)



But fortunately, we can be smarter about our growth strategy. There are strategies that will always result in either a closed tableau or at least one finite open branch. There are two aspects to a tableau-growing strategy: deciding which branch to grow and within each branch, deciding which formula (or formula-variable pair[8]) to expand.

In order to make it easier to prove things about the strategies, I will introduce the notion of an $\mathcal{RC}^\dagger$ *Hintikka set.*[9] Typically, a Hintikka set for a logic is a (potentially infinite) set of formulas for that logic that are maximally consistent in a particular way. Hintikka sets are often used as a stepping stone for proving completeness for a variety of proof systems, especially tableau systems.[10] For my purposes, the members of the Hintikka set will be pairs of a list of variables and a formula, just like the nodes of the tableau. This essentially includes formulas of $\mathcal{RC}^\dagger$ (in the form of an empty list paired with a formula), but it also includes things like $x : \forall(a, see)$ and $x, y : \overline{hate}$. For any such set of pairs H, let

[8]The existential formulas are each expanded only once, but each universal formula can be expanded for every constant variable. To keep my sentences from becoming too bloated, when the meaning is clear from context, I will sometimes write just "formulas," even though I really mean "formulas and formula-variable pairs."

[9]I'm still not sure exactly *how much* "easier" things get with Hintikka sets, but it's certainly the traditional way to do it, and I don't feel like rocking the boat on this one.

[10]See [5] for examples involving propositional and first-order logic.

vbl(H) denote the set of variables that occur within H. An $\mathcal{RC}^\dagger$ *Hintikka set* is a set H of variable-list/formula pairs satisfying the following three conditions. (Here σ is any list of variables from vbl(H).)

1. For any atom $p \in \mathbf{P} \cup \mathbf{R}$, if $\sigma : p \in H$, then $\sigma : \overline{p} \notin H$.

2. For any predicates p and q, if $\sigma : \forall(p, q) \in H$, then for any x with $x : p \in H$, it's also true that $\sigma, x : q \in H$.

3. For any predicates p and q, if $\sigma : \exists(p, q) \in H$, then there is some x such that $x : p \in H$ and $\sigma, x : q \in H$.

Lemma 3.4 (Hintikka)**.** *Any $\mathcal{RC}^\dagger$ Hintikka set is satisfiable.*

Proof. Let H be an $\mathcal{RC}^\dagger$ Hintikka set. Define the model $\mathcal{M} = \langle \text{vbl}(H), [\![\cdot]\!] \rangle$ as follows. For any variable $x \in \text{vbl}(H)$, let $[\![x]\!] := x$. For any (unary or binary) atom $p \in \mathbf{P} \cup \mathbf{R}$, let $[\![p]\!] := \{\sigma \mid \sigma : p \in H\}$.[11] By induction, I will show that every $F \in H$ is satisfied by $\mathcal{M}$.

If F is atomic, i.e., if $F = \sigma : p$ for some $p \in \mathbf{P} \cup \mathbf{R}$, then by definition, $\sigma \in [\![p]\!]$, and so $\mathcal{M} \models \sigma : p$ If F is another literal, i.e., if $F = \sigma : \overline{p}$ for some $p \in \mathbf{P} \cup \mathbf{R}$, then because H is Hintikka, we know that $\sigma : p \notin H$. In that case, we have $\sigma \notin [\![p]\!]$, and so $\mathcal{M} \models \sigma : \overline{p}$.

For the universal case, if $F = \sigma : \forall(p, q)$ for unary predicate p and (unary or binary) predicate q, I need to show that for any $x \in [\![p]\!]$, $\sigma, x \in [\![q]\!]$. So choose some $x \in [\![p]\!]$. This means that $x : p \in H$, and since H is Hintikka, $\sigma, x : q \in H$. By the induction hypothesis, $\mathcal{M} \models \sigma, x : q$, and so $\sigma, x \in [\![q]\!]$. This works for any $x \in [\![p]\!]$, and so $\mathcal{M} \models \sigma : \forall(p, q)$.

Lastly, consider the existential case, where $F = \sigma : \exists(p, q)$ for unary predicate p and (unary or binary) predicate q. Because H is Hintikka, there is an x with $x : p \in H$ and $\sigma, x : q \in H$. And so the induction hypothesis gives us $x \in [\![p]\!]$ and $\sigma, x \in [\![q]\!]$, which shows that $\mathcal{M} \models \sigma : \exists(p, q)$. □

The simplest strategy I can think of tackles the branches in a breadth-first search, and within each branch, handles the formulas in a FIFO (First In, First Out) queue. When a new existential formula appears in a branch, add it to the end of the queue. When a new universal formula appears in a branch, pair it up with all the variables that have appeared in the branch so far and add those pairs to the queue. (It doesn't really matter what order these new pairs are added, but to make the strategy deterministic, add them in the order that the variables first appeared.) When a new variable is introduced, pair it up with all the universal formulas that have appeared in the branch so far and add all those pairs to the branch. (Again, to make things deterministic, add them in the order that the formulas appeared.) Call this the *FIFO strategy.*

Since it's a deterministic strategy, we can talk about *the* FIFO-strategy tableau for a particular list of root formulas. (Sometimes I will be lazy and

[11] I'm being a little sloppy about notation here. Depending on context, σ is either a list of variables connected by commas (say, x, y) or the corresponding ordered tuple ($\langle x, y \rangle$). It should be clear from context which is intended.

talk about the FIFO-strategy tableau for a *set* of root formulas, in which case you can just assume that there's a particular order on that set.) The FIFO strategy can have infinite branches, but only if the root formulas are satisfiable, and even then, it will always have a finite open branch. The reason for this is that every open branch (this includes any infinite branches) of a tableau grown using the FIFO strategy can be used to produce a model.

Lemma 3.5 (The FIFO Strategy)**.** *Given any non-closed branch of the* FIFO*-strategy tableau for a set of root formulas Δ, there is a model $\mathcal{M} = \langle M, [\![\cdot]\!] \rangle$ that satisfies every node in the branch (in particular, $\mathcal{M} \models \Delta$).*

Proof. There's not much to the proof here. It's mostly a matter of showing that the nodes of the branch form an $\mathcal{RC}^\dagger$ Hintikka set. Since it's not closed, it clearly satisfies the first condition. Due to the first-in/first-out nature of the queue and because at every step, only finitely many formulas are added to the queue, every formula and formula-variable pair is eventually expanded in the branch. The expansion rules make sure that the second and third Hintikka conditions are also met. So the Hintikka Lemma gives us the model we need. □

If you follow the FIFO strategy, even infinite branches yield models, so every branch of a tableau with unsatisfiable root formulas must eventually be closed off. That gives us the following theorem.

Theorem 3.6 ($\mathcal{RC}^\dagger$ Completeness)**.** *Every unsatisfiable set of $\mathcal{RC}^\dagger$ formulas has a closed tableau.*

Theorems 3.3 and 3.6 together imply that the FIFO strategy can be used as an algorithm to determine the satisfiability of any set of $\mathcal{RC}^\dagger$ formulas in a finite amount of time. Simply follow the strategy and if the tableau closes off, the set is unsatisfiable. If it doesn't close off, it will eventually produce a finite, saturated branch which can be used to easily generate a finite model.

3.2 Other Strategies

The FIFO strategy is a relatively naïve strategy, but it suits the purpose of proving the existence of an algorithm for deciding $\mathcal{RC}^\dagger$ satisfiability that also provides finite models. But there are some things that can be done to speed things up, at least in some cases. There are two things we can adjust. The first is how we decide which formula to expand next in each branch (that's the FIFO part of the FIFO strategy). We can speed things up here by either attempting to get contradictions earlier (allowing us to close off branches) or by pushing the more heavily branching rules further down the tableau (creating less redundancy). The danger with adjusting this order is that you can ruin completeness (for unsatisfiability) by creating a situation where some formula never gets expanded (because it keeps getting pushed further down the queue).

In the $\mathcal{RC}^\dagger$ tableaux I've defined in this article, universal expansion rules always produce two branches, but after the first two of variables are introduced, existential rules always produce more than two branches. So we can try to eliminate some redundancy by taking care of all universal expansion steps before expanding any existential formulas. I call this the *universal-first* strategy. Fortunately, this strategy does not ruin completeness, provided we take a first-in/first-out approach once we do decide to tackle an existential formula. The reason is that universal expansion rules never introduce any new variables. There are only finitely many nodes that can ever be introduced using the finite set of formulas, predicates, and variables that already exist inside the branch.

A more complex approach is to try to close off branches as soon as possible. One way to do this is to give priority to expanding any universal formula-variable pair where one of the resulting two branches will immediately be closed off. For example, the expansion of the formula $\forall\big(p, \exists(q, see)\big)$ for the variable c would produce a branch with the node $c : \overline{p}$, so if $c : p$ is already in the branch, this expansion should be moved to the front of the queue. I call this the *immediate closure* strategy. As long as you stick to moving universal expansions up in the queue and don't move the existential formulas down in the queue, completeness is assured.

The second thing that we can adjust is the way in which we decide which branch to work on. This doesn't affect the tableau we're growing, but it can affect the order in which we find finite models, and if the tableau has infinite branches, it can change whether we find them at all. Obviously, a depth-first search is a bad idea when there might be infinite branches. But there are other options. For example, if we're interested in finding the *smallest possible* finite model, then instead of taking a pure breadth-first search pattern, we can prioritize branches based on the number of constant variables that appear in them. More precisely, we postpone growing the rightmost branches of existential expansion steps (those're the one that introduce new variables) until after we've expanded along every other branch. This means that all possible size-n models will be investigated before we even consider any size $n + q$ models. I call this a *model-size-tiered* search. Within the model-size tiers, we can use any search method, including depth-first, without worrying about getting stuck working on an infinite branch. This is because any infinite branch must necessarily contain infinitely many constant variables.

4 $\mathcal{RC}$-**tab**ModG: an Implementation

$\mathcal{RC}$-tabModG[12] (**Tab**leau-based **Mod**el **G**enerator) is my implementation of this tableau system in MiniKanren, a relational programming system embedded in Scheme.[13] One reason I chose MiniKanren was to take advantage of its **cond**^{e}operator, which effectively allows the program to investigate each branch of the tableau more or less in parallel. This sidesteps the issue of deciding which branch to grow when, which doesn't affect the tableau being grown, only in what order we grow it. This puts the focus on the other aspects of strategy, which do affect what tableau we'll end up with. The other big reason is that choosing a

[12]Pronounced "R-C-tab-modge".

[13]Check out [3] or [6] for good introductions to MiniKanren.

relational language like MiniKanren allows the program to do things other than just taking formulas and grow tableaux, such as taking existing tableaux and verifying their correctness.

$\mathcal{RC}$-tabModG is related to αleanTAP by Near, Byrd, and Friedman [12] in that it is also a tableau-based theorem prover implemented in MiniKanren, but they are put together quite differently. αleanTAP is a theorem prover for first-order logic, a descendant of leanTAP, a tableau-based theorem prover in Prolog.[1] As theorem *provers*, neither of these produce countermodels, and the difference in focus means that they take an entirely different approach to tableaux.

I won't be giving explicit details of the implementation here, but I'll describe things in general. Source code is available upon request. But a few words about MiniKanren are called for before I start talking about $\mathcal{RC}$-tabModG.

It's a tradition in MiniKanren to add the suffix $^{-o}$ to the names of relational operators (which attempt to create a relation between inputs), to distinguish them from standard Scheme operators (which have an input and an output). So while the traditional Scheme operator **car** takes a list as input and outputs the object that is at the head of the list, the MiniKanren relation **car**^{o} has two inputs (a list and an object) and tries to ensure that that the object (the second input) is the head of the list (the first input). When the first input is fully specified (meaning that we give an actual list) and the second is left unspecified (because it's a variable created by the Scheme operator **lambda** or one of the MiniKanren operator **fresh** or **run**), we say that **car**^{o} is being *run forwards.* In that case, **car**^{o} will succeed, and will identify the first member of the list with the variable given for the second input. If the first input is left unspecified, but the second input is fully specified, then we call this *running backwards*, and **car**^{o} will succeed, with the result being that the first input is now partially specified as a list with the second input as its head and an unspecified tail. If both inputs are all or partially specified, then **car**^{o} may succeed or fail, depending on whether it's possible to unify the head of the first input with the second input.

The relational operators of MiniKanren (like **car**^{o}) are typically called using the **run** operator, which attempts to make all the necessary unifications and assigns names to all the variables that remain unspecified at the end of the process. Many relations can succeed in a variety of ways. For example, the **member**^{o} relation might succeed because the second argument is the first member of the first argument, or because it's the second member of the first argument, or because it's the third... So the **run** operator takes a number as one of its arguments, indicating how many different answers are desired. There's also a **run*** version that returns a list of all possible answers, but in many underspecified cases, the process will never terminate, as MiniKanren will never run out of possibilities to try.

Most of the functionality of $\mathcal{RC}$-tabModG is in the *grow*^{o} relation, which relates a list of the formulas in a (usually partial) branch (the `branch-so-far`), a list of the formulas (and formula-variable pairs) that still need to be expanded, and a list of the variables that have already appeared to a list of the formulas that have yet be added to the branch (the `branch-to-be`). When running in the forward direction, this last list of formulas yet to be added is left unspecified by the user and is filled in by MiniKanren.

We could also specify the entire `branch-to-be` and ask it to verify that it was grown correctly according to the strategy specified. Theoretically, we could also specify all or part of the `branch-to-be` and ask MiniKanren to give us all or part of the `branch-so-far`, but it's difficult to think of good ways to make use of that in practice. Perhaps if you were trying to find a model for a set of formulas and most of the models produced were degenerate in some way, you could specify some extra properties for the `branch-to-be` to cull out undesirable branches.

Of course, there are usually many different ways of filling in the rest of the branch, and this is where the $\mathbf{cond}^e$ operator is useful. The code doesn't specify which branch will fill out the `branch-to-be`, but it does specify what formulas will appear in each of the possible branches as different $\mathbf{cond}^e$ cases. If asked for a certain number of branches, MiniKanren will keep growing branches in parallel until it has enough branches that are complete (either closed or finite and saturated (and therefore open)), or until *every* branch is complete, even if there aren't enough. If asked for *all* branches, it will keep going until every branch is complete. This is usually a bad idea, however, as many tableaux will have infinite branches, and so the process will never terminate.

Usually, the user won't interact with $grow^o$ directly, instead using either the $branch^o$ or $open\text{-}branch^o$ relations, which will fill in the unexpanded formulas and variables automatically before calling $grow^o$. In the case of $open\text{-}branch^o$, the closed branches will be filtered out. It's a simple matter to read off a satisfying model from an open branch, but the relation $model^o$ will do that for you, if you're feeling lazy. Similarly, if you'd like the program to print all or part of a tree instead of a list of branches, you can use the impure operator *grow-tableau*. Both $model^o$ and *grow-tableau* also do some cleaning up to make things more readable.[14]

4.1 Performance

Currently, I have implemented three versions of $\mathcal{RC}$-`tab`ModG, one implementing the FIFOstrategy, one implementing the universal-first (UF) strategy, and one implementing both the universal-first and the immediate-closure (UFIC) strategies. As far as I know, there are no standard sets of $\mathcal{RC}^\dagger$ formulas for testing performance, but I have compared runtimes and memory requirements for finding models for a few hand-selected sets of $\mathcal{RC}^\dagger$ formulas, some of which I have presented here. This data is from running Petite Chez Scheme on a laptop[15] running Windows 7 (64 bit), so the time comparisons are really only relevant when compared to each other. The number ("one" or "ten") represents how many models I asked $\mathcal{RC}$-`tab`ModG to generate. In the case of $\mathcal{S}_2$, there are fewer than 10 branches in the complete tree, so I just asked for all of them.

[14]The default is to name new variables after the formula which generated them, but this can be confusing to read, so these operators rename the variables using numbered indices.

[15]If you must know, it's a Samsung Notebook with a 2.20 GHz Intel i7 CPU and 6 GB RAM.

In the case of $\mathcal{R}_4$ and $\mathcal{RC}_3$, the sets are unsatisfiable, so I didn't ask for more models.

	Time			Space		
Problem	FIFO	UF	UFIC	FIFO	UF	UFIC
$\mathcal{S}_2$: one	1.20 s	1.14 s	1.25 s	155 MB	157 MB	164 MB
$\mathcal{S}_2$: all	1.92 s	2.53 s	2.701 s	284 MB	286 MB	290 MB
$\mathcal{R}_1$: one	1.78 s	1.92 s	1.98 s	270 MB	270 MB	250 MB
$\mathcal{R}_1$: ten	62.5 s	65.1 s	80.2 s	3.76 GB	3.75 GB	4.03 GB
$\mathcal{R}_4$: one	2.46 s	2.47 s	1.44 s	587 MB	512 MB	296 MB
$\mathcal{RC}_1$: one	0.764 s	0.764 s	0.842 s	169 MB	168 MB	177 MB
$\mathcal{RC}_1$: ten	7.19 s	7.57 s	7.71 s	790 MB	796 MB	843 MB
$\mathcal{RC}_3$: one	140 s	86.5 s	93.7 s	18.0 GB	10.7 GB	10.6 GB
$\mathcal{RC}_4$: one	3.40 s	3.54 s	3.92 s	770 MB	753 MB	818 MB
$\mathcal{RC}_4$: ten	37.8 s	39.2 s	40.2 s	3.13 GB	3.22 GB	3.57 GB
$\mathcal{RC}_{20}$: one	3.84 s	4.38 s	1.08 s	625 MB	633 MB	161 MB
$\mathcal{RC}_{20}$: ten	21.3 s	23.6 s	17.2 s	1.89 GB	1.89 GB	1.34 GB
$\mathcal{RC}_{21}$: one	8.29 s	5.51 s	0.764 s	998 MB	900 MB	133 MB
$\mathcal{RC}_{21}$: ten	191 s	200 s	11.3 s	15.1 GB	14.0 GB	1.08 GB

For small sets of relatively simple formulas, the overhead involved in using the more complex strategies outweighs any benefit gleaned from smaller trees. When the formulas become more complex, and especially if there are many closed branches, the universal-first/immediate-closure strategy becomes the clear winner, outshining the others by orders of magnitude. For some particular sets of formulas, the universal-first strategy is faster than the others, but the gains are modest in those cases. Once the sets of formulas become complex enough, the universal-first/immediate-closure strategy becomes a clear winner. Runtimes for unsatisfiable sets can quickly spiral out of control, and a couple carefully chosen formulas (see example $\mathcal{RC}_6$) with quadruply nested quantifiers can explode the runtimes beyond my patience to measure (more than 24 hours).

5 Conclusion

In this paper, I've presented an analytic tableau system for reasoning about the relational syllogistic logic $\mathcal{RC}^\dagger$ and its sublogics, with a focus on model generation. The logic is sound and complete, but it is not *strongly* complete in general (bad strategies can result in infinite tableaux, even for unsatisfiable formulas). It *is* strongly complete for the sublogics $\mathcal{S}$ and $\mathcal{S}^\dagger$. For $\mathcal{RC}^\dagger$, I've presented several strategies which will always produce a finite model (if the root formulas are satisfiable) or a closed tableau (if they aren't) in a finite amount of time. I've implemented the tableau system in MiniKanren (a relational programming system embedded in Scheme).

But my work here is not done! There are a number of improvements that can be made to the implementation to make it more efficient and more user-friendly. I would also like to investigate extending the tableau system to more complex syllogistic logics, in particular, for $\mathcal{RCA}^\dagger$, which extends $\mathcal{RC}^\dagger$ by adding

Figure 3: Sample problems

$$\begin{gathered}\forall(a,b)\\ \forall(b,c)\\ \forall(c,d)\\ \forall(a,\overline{d})\\ \exists(d,e)\end{gathered}$$

$\mathcal{S}_2$: satisfiable

$$\begin{gathered}\forall\big(a,\forall(c,s)\big)\\ \forall\big(a,\exists(d,s)\big)\\ \exists(d,b)\\ \forall\big(b,\forall(d,s)\big)\\ \forall(b,\overline{c})\end{gathered}$$

$\mathcal{R}_1$: satisfiable

$$\begin{gathered}\forall\big(a,\forall(b,s)\big)\\ \forall\big(a,\forall(b,\overline{s})\big)\\ \exists(b,b)\\ \exists(a,a)\end{gathered}$$

$\mathcal{R}_4$: unsatisfiable

$$\begin{gathered}\forall(k,m)\\ \exists\big(\forall(k,s),\exists(m,\overline{s})\big)\end{gathered}$$

$\mathcal{RC}_1$: satisfiable

$$\begin{gathered}\forall(k,m)\\ \exists\Big(\forall\big(\forall(k,h),s\big),\exists\big(\forall(m,h),\overline{s}\big)\Big)\end{gathered}$$

$\mathcal{RC}_3$: unsatisfiable

$$\begin{gathered}\forall(k,m)\\ \exists\Big(\forall\big(\exists(k,h),s\big),\exists\big(\exists(m,h),\overline{s}\big)\Big)\end{gathered}$$

$\mathcal{RC}_4$: satisfiable

$$\begin{gathered}\forall(k,m)\\ \exists\bigg(\forall\Big(\forall\big(\exists(k,h),s\big),r\Big),\exists\Big(\exists\big(\exists(m,h),s\big),\overline{r}\Big)\bigg)\end{gathered}$$

$\mathcal{RC}_6$: unsatisfiable

$$\begin{gathered}\forall\big(\exists(d,r),\exists(c,s)\big)\\ \forall\big(a,\forall(c,r)\big)\\ \forall\big(\exists(c,r),b\big)\\ \exists\big(b,\forall(e,r)\big)\\ \forall(e,d)\end{gathered}$$

$\mathcal{RC}_{20}$: satisfiable

$$\begin{gathered}\forall\big(\exists(a,r),\exists(b,r)\big)\\ \exists\big(c,\forall(a,r)\big)\\ \forall\Big(c,\exists\big(\exists(a,s),s\big)\Big)\end{gathered}$$

$\mathcal{RC}_{21}$: satisfiable

comparative adjectives, which are automatically transitive and reflexive. $\mathcal{RCA}^{\dagger}$ does not have the finite model property, and I would like to investigate tableau-based strategies for identifying repeating patterns in infinite branches with the intent of finding infinite models in a finite amount of time.

References

[1] Bernhard Beckert and Joachim Posegga. leantap: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[2] George Boolos. Trees and finite satisfiability: proof of a conjecture of burgess. *Notre Dame Journal of Formal Logic*, 25(3):193–197, 1984.

[3] William E Byrd. *Relational programming in MiniKanren: techniques, applications, and implementations.* PhD thesis, Indiana University Bloomington, 2010.

[4] Johannes Dellert. Challenges of model generation for natural language processing. Master's thesis, University of Tübingen, 2011.

[5] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Graduate Texts in Computer Science. Springer, second edition, 1996.

[6] Daniel P Friedman, William E Byrd, and Oleg Kiselyov. *The Reasoned Schemer.* The MIT Press, second edition, 2006.

[7] Erich Grädel, Phokion G Kolaitis, and Moshe Y Vardi. On the decision problem for two-variable first-order logic. *Bulletin of symbolic logic*, pages 53–69, 1997.

[8] Reiner Hahnle. Tableaux and related methods. *Handbook of automated reasoning*, 1:101–178, 2001.

[9] David A McAllester and Robert Givan. Natural language syntax and first-order inference. *Artificial Intelligence*, 56(1):1–20, 1992.

[10] Michael Mortimer. On languages with two variables. *Mathematical Logic Quarterly*, 21(1):135–140, 1975.

[11] Reinhard Muskens. An analytic tableau system for natural logic. In *Logic, Language and Meaning*, pages 104–113. Springer, 2010.

[12] Joseph P Near, William E Byrd, and Daniel P Friedman. αleantap: A declarative theorem prover for first-order classical logic. In *Logic Programming*, pages 238–252. Springer, 2008.

[13] Ian Pratt-Hartmann and Lawrence S Moss. Logics for the relational syllogistic. *The Review of Symbolic Logic*, 2(04):647–683, 2009.