Two Advances in the Implementations of Extended Syllogistic Logics

Jason Hemann, Cameron Swords, and Lawrence S. Moss
{jhemann, cswords, lmoss}@indiana.edu

Indiana University, Bloomington

Abstract. Natural logics are of interest to both logicians and members of the natural-language research community. They provide a means of precisely reasoning about aspects of natural language in a way that is computationally tractable, and akin to the process by which humans reason in ordinary language. This paper takes as a target a reasonably-small logic which support reasoning about "All", "Some", negated nouns, relative clauses, and the "more X than Y" operation of cardinality comparison. The importance of this logic is that it goes beyond first-order logic, hence one cannot use off-the-shelf tools. This paper contains two contributions to the implementation of this logic and others. First, it mentions an implementation in Sage. The program builds proofs and counter-models by one and the same algorithm. That is, the failure to build a proof provides the data for a counter-model in an automatic way. This abstract does not go into details on the algorithm, but a talk on this includes a demo of the Sage program. Second, in a very different direction, we mention *declarative implementations* of a different logic in this family, done in the miniKanren language. These implementations provide users with automated proof search, theorem generation, and proof checking, and are designed to facilitate reuse in implementing other natural logics.

1 Introduction

Logical syllogisms—arguments with deductive reasoning—have been the object of study since at least Aristotle. More recently, these logical syllogisms become the core of a series of *natural logics* [14], which aim to mirror the style of deductions people employ in everyday reasoning.

Though well-studied, there has been little work toward developing automated tools for proof searches in natural logics. In the applications work that *has* been undertaken [22,19,18,17,4,12,27,25], the implementations themselves have not been the object of study; implementers have been content to use imperative implementations, or rely on SAT solvers or tools like Prover9 and Sage. These can provide powerful, performant tools for working with these logics. But, they somewhat obscure the direct nature of the reasoning employed. A high-level, declarative implementation of these logics, on the other hand, preserves and

highlights the direct reasoning of these deduction systems in their implementations. This declarative approach also provides to be extensible and well-suited for rapid prototyping.

We utilize the declarative language miniKanren to demonstrate this encoding process for several logics, including a new logic for cardinality comparison of atoms, from their proof-tree derivation rules. For each logic, we produce for each a single tool that can be used for proof instantiation, proof derivation, and automated theorem search from a list of premises. The full code may be found at `http://github.com/jasonhemann/natlogic`. This paper proceeds as follows:

- Section 2 discusses the logic with cardinality comparison, and it shows examples of the Sage implementation.
- Section 3 serves as a brief primer to the miniKanren language and the cKanren implementation, embedded in Racket. It also describes the basic strategy to encode natural logics as miniKanren programs, including premise representation, proof construction, and user invocation. We demonstrate this approach by encoding $\mathcal{A}$, the logic of 'All", in miniKanren.
- Section 4 describes a miniKanren implementation of a logic with cardinality comparison.
- Section 5 discusses related work and concludes and describes potential future work.

2 A Logic for Cardinality Comparison

We begin by introducing a logic for cardinality comparison on top of the basic syllogistic logic, taken from [20]. Consider the following argument:

$$\frac{\begin{array}{l}\textsf{There are more students than professors at the party}\\ \textsf{There are more professors than deans at the party}\end{array}}{\textsf{There are more students than deans at the party}} \quad (1)$$

The conclusion follows from the premises. The intuition is that the transitivity of more ... than ... is a basic feature of human reasoning, on a par with the transitivity of all ... are ... that we see in the syllogistic rule (BARBARA). We do not wish to formalize the argument in (1) by translating it into another logic (for example, logical systems which incorporate natural numbers); the point is that the general logical principles of the target systems are likely to be much more complicated than necessary for this task.

Let us widen the discussion a little. In addition to more ... than ..., we also find in language the weaker assertion there are at least as many ... as .... Here is another argument which we take to be valid:

$$\frac{\begin{array}{l}\textsf{There are at least as many rabbits as deer}\\ \textsf{There are more deer than goats}\end{array}}{\textsf{There are more rabbits than goats}} \quad (2)$$

$$\frac{}{\forall(p,p)}\ (\textsc{axiom}) \qquad \frac{\forall(n,p) \quad \forall(p,q)}{\forall(n,q)}\ (\textsc{barbara})$$

$$\frac{\exists(p,q)}{\exists(p,p)}\ (\textsc{some}) \qquad \frac{\exists(q,p)}{\exists(p,q)}\ (\textsc{conversion})$$

$$\frac{\exists(p,n) \quad \forall(n,q)}{\exists(p,q)}\ (\textsc{darii}) \qquad \frac{\forall(p,q) \quad \exists^{\geq}(p,q)}{\forall(q,p)}\ (\textsc{card-mix})$$

$$\frac{\forall(p,q)}{\exists^{\geq}(q,p)}\ (\textsc{subset-size}) \qquad \frac{\exists^{\geq}(n,p) \quad \exists^{\geq}(p,q)}{\exists^{\geq}(n,q)}\ (\textsc{card-trans})$$

$$\frac{\exists(p,p) \quad \exists^{\geq}(q,p)}{\exists(q,q)}\ (\textsc{card-}\exists) \qquad \frac{\exists^{>}(p,q)}{\exists^{\geq}(p,q)}\ (\textsc{more-at least})$$

$$\frac{\exists^{>}(n,p) \quad \exists^{\geq}(p,q)}{\exists^{>}(n,q)}\ (\textsc{more-left}) \qquad \frac{\exists^{\geq}(n,p) \quad \exists^{>}(p,q)}{\exists^{>}(n,q)}\ (\textsc{more-right})$$

$$\frac{\exists^{\geq}(p,q) \quad \exists^{>}(q,p)}{\phi}\ (\mathrm{X})$$

$$\frac{\forall(p,\overline{p})}{\forall(p,q)}\ (\textsc{zero}) \qquad \frac{\forall(\overline{p},p)}{\forall(q,p)}\ (\textsc{one})$$

$$\frac{\forall(q,p) \quad \exists(p,\overline{q})}{\exists^{>}(p,q)}\ (\textsc{more}) \qquad \frac{\exists^{>}(p,q)}{\exists(p,\overline{q})}\ (\textsc{more-some})$$

$$\frac{\exists^{>}(q,p)}{\exists^{>}(\overline{p},\overline{q})}\ (\textsc{more-anti}) \qquad \frac{\forall(p,q)}{\forall(\overline{q},\overline{p})}\ (\textsc{anti})$$

$$\frac{\exists^{\geq}(p,q)}{\exists^{\geq}(\overline{q},\overline{p})}\ (\textsc{card-anti}) \qquad \frac{\exists(p,p) \quad \exists^{\geq}(q,\overline{q})}{\exists(q,q)}\ (\textsc{int})$$

$$\frac{\exists^{\geq}(p,\overline{p}) \quad \exists^{\geq}(\overline{q},q)}{\exists^{\geq}(p,q)}\ (\textsc{half}) \qquad \frac{\exists^{>}(p,\overline{p}) \quad \exists^{\geq}(\overline{q},q)}{\exists^{>}(p,q)}\ (\textsc{strict half})$$

$$\frac{\exists^{\geq}(p,\overline{p}) \quad \exists^{\geq}(q,\overline{q}) \quad \exists(\overline{p},\overline{q})}{\exists(p,q)}\ (\textsc{maj})$$

Fig. 1. Rules for the cardinality logic. The rules for a smaller system, which lacks complemented variables, are found above the line.

And here is an argument of a different character:

$$\frac{\begin{array}{l}\mathsf{All\ violas\ are\ stringed\ instruments}\\ \mathsf{There\ are\ at\ least\ as\ many\ violas\ as\ stringed\ instruments}\end{array}}{\mathsf{All\ stringed\ instruments\ are\ violas}} \tag{3}$$

A moment's thought will convince the reader that this is valid, provided that we are speaking of *finite* situations. In this paper we restrict attention to finite universes, in order to obtain a logical system that we think of greater "human interest" than the weaker logic that would result if we allowed infinite structures and thus denied the validity of (3).

Finally, we make our logical language more expressive by allowing *complementation of nouns*. Here are some examples:

$$\frac{\mathsf{There\ are\ at\ least\ as\ many}\ x\ \mathsf{as}\ y}{\mathsf{There\ are\ at\ least\ as\ many\ non\text{-}}y\ \mathsf{as\ non\text{-}}x} \qquad \frac{\begin{array}{l}\mathsf{There\ are\ at\ least\ as\ many}\ x\ \mathsf{as\ non\text{-}}x\\ \mathsf{There\ are\ at\ least\ as\ many}\ y\ \mathsf{as\ non\text{-}}y\end{array}}{\mathsf{There\ are\ at\ least\ as\ many}\ x\ \mathsf{as\ non\text{-}}y} \tag{4}$$

The first example just above shows an inference whose soundness depends on the fact that we are looking at a finite universe. The second uses a property of "half": if the universe has N objects, the premises tell us that the xs are at least $\frac{N}{2}$ in number. The ys number at least $\frac{N}{2}$, and so the non-ys number at most $\frac{N}{2}$. Thus the xs number at least as much as the non-ys. The fact that we can do all of this with cardinality comparison and complement makes this work interesting and non-trivial.

The main result in [20] is a sound and complete logical system whose sentences are of the form $\mathsf{All}\ x\ \mathsf{are}\ y$, $\mathsf{Some}\ x\ \mathsf{are}\ y$, $\mathsf{There\ are\ at\ least\ as\ many}\ x\ \mathsf{as}\ y$, and $\mathsf{There\ are\ more}\ x\ \mathsf{than}\ y$. Moreover, th logic does not involve translating the cardinality assertions into any other language. The proof system is sound and strongly complete: for a finite set $\Gamma \cup \{\phi\}$ of sentences, ϕ is true in every model of Γ if and only if there is a derivation of ϕ from Γ. This paper does not discuss the completeness result at all but rather presents the implementation.

Formal System. Figure 1 presents the rules of the system in natural-deduction format. The logic has sentences of the following forms:

- $\forall(p, q)$ read as "all p are q",
- $\exists(p, q)$ read as "Some p are q",
- $\exists^{\geq}(p, q)$ read as "there are at least as many p as q",
- $\exists^{>}(p, q)$ read as "there are more p than q"

There are no connectives, and the overline symbol ($\overline{p}$) on the variables is for set complement. This logic induces a notion of models and precise definitions of model satisfaction such that we may define what it means for a (finite) set of sentences to semantically imply another sentence.

Implementation. On the other hand, this paper is about the implementation. The logical consequence has been implemented in Sage, and the implementation is currently available on `https://cloud.sagemath.com`. (That is, it can be shared.) The consequence relation may be computed in polynomial time. This should be a little surprising, since the cardinality comparison machinery cannot be expressed in first-order logic.

Example 1. One may enter:

```
 assumptions= ['All non-a are b',
               'There are more c than non-b',
               'There are more non-c than non-b',
               'There are at least as many non-d as d',
               'There are at least as many c as non-c',
               'There are at least as many non-d as non-a']
conclusion = 'All a are non-c'
follows(assumptions,conclusion)
```

The last line indicates that we are asking if a given conclusion follows from a given list of six assumptions. Then the program returns, telling us that the conclusion does not follow. And it produces a *counter-model*, a model where all of the assumptions are true and the conclusion false.

```
Here is a counter-model.
We take the universe of the model to be {0, 1, 2, 3, 4, 5}

  noun   semantics      complement
+------+--------------+--------------------+
  a      {2, 3}         {0, 1, 4, 5}
  b      {0, 1, 4, 5}   {2, 3}
  c      {0, 2, 3}      {1, 4, 5}
  d      {}             {0, 1, 2, 3, 4, 5}
```

So it gives the semantics of `a`, `b`, `c`, and `d` as subsets of $\{0, \ldots, 5\}$. Notice that the assumptions are true in the model, but the conclusion is false. In the cases that the conclusion did follow, the system would output a proof in our system.

Example 2. Here is an example of a derivation found by our implementation. We ask whether the putative conclusion below really follows:

$$\frac{\begin{array}{l}\text{All non-x are x}\\ \text{Some non-y are z}\end{array}}{\text{There are more x than y}}$$

The program returns the following result when we provide it the assumptions listed above, asking for a profe that there are more x than y:

```
1   All             non-x   are    x         Assumption
2   All             y       are    x         One   1
3   All             non-x   are    x         Assumption
4   All             non-y   are    x         One   3
5   Some            non-y   are    z         Assumption
6   Some            non-y   are    non-y     Some  5
7   Some            non-y   are    x         Darii   4   6
8   Some            x       are    non-y     Conversion   7
9   There are more  x       than   y         More   2   8
```

While the proof is displayed as a list rather than a tree, it is merely a cosmetic difference.

The advantage of working with a syllogistic system formulated using *ex falso quodlibet* rather than *reductio ad absurdum* is that the proof search and the counter-model generation are closely related. In a sense, they are both results of the same algorithm. Moreover, the algorithm is efficient. That is, the question of whether a sentence follows from a list of assumptions is in polynomial time.

Unfortunately, the implementation is obscuring: it is over 1500 lines of Sage and ultimately relies on iteratively constructing *all possible derivations* from a given set of assumptions. Moreover, this implementation is customized toward dealing with the logic of relative cardinalities. These features are hardly ideal when experimenting with a new logic.

To this end, we now turn our focus to miniKanren, a declarative programming language embedded in Racket, to demonstrate how it can be used to develop proof searches for natural logics. Unlike our Sage work, the miniKanren implementation of this cardinality logic is concise, clear, and extensible. It is, however, unable to generate counter-models and takes super-polynomial time: the miniKanren approach is good for experimentation, but has sub-par performance.

3 Implementations in miniKanren

Our previous section discussed a Sage implementation of a single large logic with distinct advantages in disadvantages. In its favor, the algorithm works in polynomial time and includes counter-model generation along with proof search. Unfortunately, the algorithm is highly specialized toward the logics and difficult to explain (and thus eschewed in our presentation). In a different direction, we present a generic way to build *declarative implementations* of syllogistic logics. The key point is the genericity of the work: it is straightforward to construct and experiment with proof searches for these logics in a declarative language. On the other hand, these constructions are not as closely related to counter-model search and the resultant encodings are less efficient than custom-constructed algorithms.

3.1 miniKanren: A Brief Introduction

miniKanren is a family of embedded, domain-specific relational (logic) programming languages [3,5,6,2,10,9]. Implementations come with a variety of constraints, the foremost of which is `==`, an equality constraint implemented with syntactic, first order unification. For this presentation, we use cKanren, an implementation of miniKanren embedded in Racket [7]. The cKanren implementation provides programmers with access to the entirety of the host language when writing miniKanren programs[1] and the ability to define their own constraints.

The `run` Interface. The primary interface to miniKanren is `run`, which takes the *maximal* number of answers desired, an "output" variable—a variable with respect to which the answer should be presented—and a sequence of goal expressions to achieve. Consider the following miniKanren program execution:

```
> (run 1 (q) (== q 3))
'(3)
```

Here, the `1` indicates we request at most one answer for the variable `q`. The only constraint is the equality of `q` with `3`. The output is always presented as a list of results; this is the list contains only one result for `q`, the value `3`. Consider this second execution:

```
> (run 1 (q) (== 3 3))
'(_.0)
```

The list of results again contains only one element, this time `_.0`. The final substitution for this program has no information regarding `q`, it is a *fresh variable*. In the presentation of the answer, distinct fresh variables are written `_.`n, where n is an index beginning at 0.

Getting `fresh` Variables. It is often useful to introduce auxiliary logic variables as a part of writing a miniKanren program. In the example below, we wish to assert the query variable is a pair; we introduce new variables `a` and `d` and use them in a constraint:

```
> (run 1 (q) (fresh (a b) (== q `(,a . ,b))))
'((_.0 . _.1))
```

The `fresh` operator takes a list of identifiers and a sequence of goal expressions over which new variables are scoped. In this case, they are scoped over a constraint equating `q` with a pair whose first element is the variable `a` and whose second is the variable `b`. We rely on the host language's term constructors to build miniKanren terms, destructuring must be performed with `==`. New variables are lexically scoped, so inner bindings shadows outer ones.

[1] Except vectors, which are used in the implementation.

Using `conde` *for Non-deterministic Computation.* The `conde` operator implements a complete search (whose details are unimportant here) that allows us to simulate a form of nondeterministic choice. It takes any number of clauses (lists of goal expressions) and operates as though each clause were attempted independently.

```
> (run 3 (q) (conde
              ((fresh (a b) (== q `(,a . ,b))))
              ((== q 6))))
'(6 (_.0 . _.1))
```

We request 3 results, but receive only two: one from each `conde` clause. miniKanren interleaves the search for results, and we are in general not guaranteed to receive the results in the order of their `conde` clauses.

Disequality Constraints. The miniKanren operator `=/=` implements disequality constraints. Placing a disequality constraint on two terms already identical in the current substitution causes failure, and if `u` and `v` are under a disequality constraint, then a substitution extension that forces `u` and `v` to be syntactically identical will also cause failure. Like the substitution, disequality constraints are carried as part of the state and are indicated in the output:

```
> (run 1 (q) (=/= q 3))
'((_.0 (=/= ((_.0 3)))))
```

The variable `q` still has no binding in the ultimate substitution, and so it is again presented as `_.0`, but we also mandate that `_.0` not be 3. Our disequality constraints, like the `dif/2` of various Prologs, fail only when their arguments are identical relative to the current substitution.

User-defined Constraints. We demonstrate an example of a user-defined cKanren constraint below. We provide a name, and specify its criteria for satisfaction and its interactions with other constraints. As part of the implementations we provide a suite of pre-built constraints for defining these and other natural logics.

The `un-atom` constraint mandates that the term be a unary atom, which for our purposes means a plural noun (e.g. "logicians"). We represent them as symbols, and we require they not overlap with binary atoms (transitive verbs).

```
(define-attribute un-atom
  #:satisfied-when symbol?
  #:incompatible-attributes (number bin-literal bin-atom))
```

Adding constraints for atoms, literals, negated literals, etc., makes the resulting answers more legible and also groups together multiple answers by collapsing the search space.

3.2 Putting Things Together

A miniKanren program attempts to satisfy a number of *goals* in a given *state*, which either *succeed*, returning a *stream* of one or more achieving states, or *fail*, yielding an empty stream. Because cKanren is embedded in Racket, miniKanren programmers have access to the entirety of Racket when writing miniKanren programs. As a result, relations can be defined globally and then invoked elsewhere, as in the following example:

```
> (define (membero x l)
    (fresh (a d)
      (== l `(,a . ,d))
      (conde
        ((== x a))
        ((=/= x a) (membero x d)))))
```

We globally define the binary relation `membero`, which holds when `x` is an elements of a list `l`. In the program below, we demand `q` be such a list containing `'x`, and we request three such elements.

```
> (run 3 (q) (membero 'x q))
'((x . _.0)
  ((_.0 x . _.1) (=/= ((_.0 x))))
  ((_.0 _.1 x . _.2) (=/= ((_.0 x)) ((_.1 x)))))
```

Furthermore, we can use Racket's macro system to extend miniKanren with additional syntactic operations, such as `matche`, a pattern matcher that will perform automatic fresh variable creation [13]. For example, consider the following two definitions of relational append:

```
(define (appendo ls1 ls2 lout)
  (conde
    ((== ls1 '())
    (== ls2 lout))
    ((fresh (a d r)
      (== ls1 `(,a . ,d))
      (== lout `(,a . ,r))
      (appendo d ls2 r)))))
```

```
(define (appendo ls1 ls2 lout)
  (matche ls1
    (()
     (== ls2 lout))
    ((,a . ,d)
     (fresh (r)
       (== lout `(,a . ,r))
       (appendo d ls2 r)))))
```

The left one creates a number of fresh variables and performs unification against `ls1` at each step. The right one performs the same operations with the pattern matching tool `matche`, dispatching on the shape of `ls1`. This approach allows us to avoid creating a number of additional variables and elide the unifications against `ls1` in the program[2]. This style of match-and-dispatch will prove invaluable in rapidly constructing logical proof search.

[2] These equations and fresh variables are created during macro expansion.

3.3 A System for Logical Encoding

With this basic understanding of natural logics and miniKanren, we now proceed with encoding natural logics in a generic and extensible way. We begin with $\mathcal{A}$, the logic of "All" relations [17,19,22,8], to demonstrate the general encoding process. $\mathcal{A}$ uses three judgement rules: environmental lookup, (AXIOM), and (BARBARA) from Figure 1. We use Γ to represent the set of premises.

These rules indicate that every object, or *unary atom*, p, is reflexively self-contained, and the containment relation is transitive. To encode these rules in miniKanren, we must build a relation that takes as arguments a theorem `φ` to prove, some environment of premises, `Γ`, and, because we are writing a relation, some proof tree output `proof`. The resultant procedure, presented in Figure 2, uses miniKanren's `==`, `conde`, `matche`, and `fresh` as well as the aforementioned parts of the host language.

```
1  (define (A φ Γ proof)
2    (matche φ
3      [(∀ ,a ,a) (== φ proof)] ;; Axiom
4      [,x (membero x Γ) (== proof `(,x in-Γ))]  ;; Lookup
5      [(∀ ,n ,q) ;; Barbara
6       (fresh (p prim1 proof1 prim2 proof2)
7         (== `((,proof1 ,proof2) => ,φ) proof)
8         (== prim1 `(∀ ,n ,p))
9         (== prim2 `(∀ ,p ,q))
10        (A prim1 Γ proof1)
11        (A prim2 Γ proof2))]))
```

Fig. 2. A miniKanren implementation for $\mathcal{A}$.

Our implementation operates over not-quite-English: like McAllester and Givan [15], we find it convenient to encode the premises provided as lists. We encode "All" sentences as `(∀ p q)` instead of McAllester and Givan's `(All p q)` structure: Racket supports unicode so we can more closely match the format of the logical rules.

This procedure is the entire encoding of $\mathcal{A}$. We begin by matching against the input `φ` and proceeding with three possibilities (one for each rule):

- The first, at line 3, asks if `φ` will unify with `(∀ ,a ,a)`—if we are stating that "All a are a." In this case, the proof follows trivially (by (AXIOM)), and thus we unify the statement with the proof tree output.
- The second, at line 4, matches generically against any `φ` and then checks if that `φ` is a member of `Γ`. If it is, we unify the proof tree with a list denoting the entailment.
- The third, on lines 6–11, encodes the transitivity rule (BARBARA) of $\mathcal{A}$. We introduce five fresh variables:

- `p`, the intermediary atom in the term
- `prim1` and `prim2`, which represent `(∀ ,n ,p)` and `(∀ ,p ,q)` respectively
- `proof1` and `proof2`, which indicate the proof terms for `(∀ ,n ,p)` and `(∀ ,p ,q)` respectively

Finally, we invoke `A` to recursively build proofs for `(∀ ,n ,p)` and `(∀ ,p ,q)`, passing in the appropriate proof variables in each case.

4 Cardinality Logic in miniKanren

With these tools in mind, we implement the cardinality object from Figure 1 in miniKanren. The first half (above the line in Figure 1) is given in Figure 3.

Similar to the preceding examples, the Racket function `card` implements a miniKanren relation that describes when that relationship holds between its inputs, and each `matche` clause of the `card` relation corresponds to a rule of Figure 1.

For larger logics such `card`, the repetition inherent in specifying the rules of the logic may become tedious: each two-premise recursion mirrors our implementation of (BARBARA) in Figure 2, and the one-premise rules follow similarly. We use Racket's macro system to once again simplify our task, creating two additional syntactic forms that construct the appropriate miniKanren terms. These new syntactic forms, `single-prim-term` and `double-prim-term` in Figure 3, take the logic's name, the environment, the term(s) that is required to hold in order for `φ` to hold, and any auxillary variables required, and use them to construct the fresh variable creation, unifications, and recursions necessary to implement the rule. For example, consider our equivalent implementations of (BARBARA) side-by-side:

```
[(∀ ,n ,q) ;; Barbara
 (fresh (p prim1 proof1 prim2 proof2)
   (== `((,proof1 ,proof2) => ,φ) proof)
   (== prim1 `(∀ ,n ,p))
   (== prim2 `(∀ ,p ,q))
   (A prim1 Γ proof1)
   (A prim2 Γ proof2))]
```

```
[(∀, n, q) ;; Barbara
 (double-prim-term
   card proof φ Γ
   `(∀ ,n ,p)
   `(∀ ,p ,q)
   p)]
```

Using these syntactic abstractions, our program is reduced to a series of pattern-matching clauses whose the left-hand sides are the translations of the conclusions of a judgment rule, and whose right-hand sides are invocations of `single-prim-term` or `double-prim-term`, encoding the antecedent or antecedents. This direct correspondence between the logical rules and the implementation facilitates rapid prototyping and quick experimentation when working with natural logics.

5 Conclusions and Next Steps

Interest in the history of syllogistic logics motivated the development of a great variety of tools (e.g., Glashof [11]). In particular, the work in Prolog has focused

```
(define-syntax double-prim-term
  (syntax-rules ()
     [(_ logic proof φ Γ e1 e2 vars ...)
      (fresh (vars ... prim1 prim2 proof1 proof2)
        (== `((,proof1 ,proof2) => ,φ) proof)
        (== prim1 e1)
        (== prim2 e2)
        (logic prim1 Γ proof1)
        (logic prim2 Γ proof2))]))

(define-syntax single-prim-term
  (syntax-rules ()
     [(_ logic proof φ Γ e1 vars ...)
      (fresh (vars ... prim1 proof1)
        (== `((,proof1) => ,φ) proof)
        (== prim1 e1)
        (logic prim1 Γ proof1))]))

(define (card φ Γ proof)
  (matche φ
    [(∀ ,a ,a) (== φ proof)] ;; Axiom
    [(∀, n, q)
     (double-prim-term card proof φ Γ `(∀ ,n ,p) `(∀ ,p ,q) p)] ;; Barbara
    [(∃ ,p ,p) ;; ∃
     (single-prim-term card proof φ Γ `(∃ ,p ,q) q)]
    [(∃ ,p ,q) ;; Conversion
     (single-prim-term card proof φ Γ `(∃ ,q ,p))]
    [(∃ ,p ,q) ;; Darii
     (double-prim-term card proof φ Γ `(∃ ,p ,n) `(∀ ,n ,q) n)]
    [(∀ ,q ,p) ;; Card-Mix
     (double-prim-term card proof φ Γ `(∀ ,p ,q) `(∃≥ ,p ,q))]
    [(∃≥ ,q ,p) ;; Subset-Size
     (single-prim-term card proof φ Γ `(∀ ,p ,q))]
    [(∃≥ ,n ,q) ;; Card-Trans
     (double-prim-term card proof φ Γ `(∃≥ ,n ,p) `(∃≥ ,p ,q) p)]
    [(∃ ,q ,q) ;; Card-E
     (double-prim-term card proof φ Γ `(∃ ,p ,p) `(∃≥ ,q ,p) p)]
    [(∃≥ ,p ,q) ;; More-At-Last
     (single-prim-term card proof φ Γ `(∃> ,p ,q))]
    [(∃> ,n ,q) ;; More-Left
     (double-prim-term card proof φ Γ `(∃> ,n ,p) `(∃≥ ,p ,q) p)]
    [(∃> ,n ,q) ;; More-Right
     (double-prim-term card proof φ Γ `(∃≥ ,n ,p) `(∃> ,p ,q) p)]
    [,x (membero x Γ) (== proof `(,x in-Γ))]  ;; Lookup
    [,x ;; X
     (double-prim-term card proof φ Γ `(∃≥ ,p ,q) `(∃≥ ,q ,p) p q)]))
```

Fig. 3. miniKanren implementation of the positive portion of the cardinality logic in Figure 1.

on natural language processing and the classical syllogistic logics (typically no further than $\mathcal{S}^{\dagger}$) [23,16,26,22,19,18,4,12,27,25]. There are a variety of such results, and it would be difficult to thoroughly catalog all of these systems here.

Our work with Sage shows that it is possible to do proof search and countermodel generation at the same time. The key point is that *reductio ad absurdum* is a derived rule, not a basic feature of the system. This is what is behind our polynomial-time algorithm.

The relational nature of miniKanren facilitates proof verification and proof search in the same implementation, and the generic style of implementation allows us to freely explore new extensions to the syntax and proof theory with little to no additional overhead. By default, miniKanren relies on a kind of breadth-first search strategy. Modifying these implementations, using techniques pioneered in rKanren [24], will allow the user to more finely tune the direction of the proof search. Additionally, future improvements in cKanren's set constraint architecture will likely enable an increase in both performance and clarity of our implementations.

But the most important next step in this line of work is to connect with the tableau system in [1] (based on [21]). Abzianidze's paper shows that computational systems based on natural logics can be combined with CCG parsers and other NLP tools in order to scale up work in this area. Indeed, he succeeds in handling RTE-like data. Nevertheless, his approach is based on tableaux, and ours is based on the complementary technique of formal proofs. So connecting the two approaches is the most important task on the road toward using natural logic in NLP.

References

1. Lasha Abzianidze. A tableau prover for natural logic and language. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 2015.
2. Claire E Alvis, Jeremiah J Willcock, Kyle M Carter, William E Byrd, and Daniel P Friedman. cKanren: miniKanren with constraints. *Scheme and Functional Programming*, 2011.
3. William E. Bird. minikanren.org. `http://minikanren.org/`. Accessed 1/18/2014.
4. Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language: A First Course in Computational Semantics (Studies in Computational Linguistics)*. Center for the Study of Language and Information, 2005.
5. William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Indiana University, 2009.
6. William E Byrd, Eric Holk, and Daniel P Friedman. minikanren, live and untagged. *Scheme and Functional Programming*.
7. Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.
8. Nissim Francez and Roy Dyckhoff. Proof-theoretic semantics for a fragment of natural language. *Linguistics and Philosophy*, 33(6):447–477, 2011.
9. Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, July 2005.

10. Daniel P. Friedman and Oleg Kiselyov. A declarative application logic programming system, 2005.
11. Klaus Glashof. Computational aristotelian term logic, 2004. see http://webapp5.rrz.uni-hamburg.de/syllogism/aristotelianlogic/.
12. Nikolay Ivanov and Dimiter Vakarelov. A system of relational syllogistic incorporating full boolean reasoning. *Journal of Logic, Language, and Information*, 21(4):433–459, 2012.
13. Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd, and Daniel P. Friedman. A pattern matcher for miniKanren or how to get into trouble with CPS macros. In *Scheme '09: Proceedings of the 2009 Scheme and Functional Programming Workshop*, number CPSLO-CSC-09-03 in California Polytechnic State University Technical Report, pages 37–45, 2009.
14. George Lakoff. Linguistics and natural logic. *Synthese*, 22:151–271, 1970.
15. David A. McAllester and Robert Givan. Natural language syntax and first-order inference. *Artificial Intelligence*, 56:1–20, 1992.
16. M McCord. Using slots and modifiers in logic grammars for natural language. *Artificial Intelligence*, 18(3):327–367, May 1982.
17. Lawrence S. Moss. Completeness theorems for syllogistic fragments. In F. Hamm and S. Kepser, editors, *Logics for Linguistic Structures*, pages 143–173. Mouton de Gruyter, 2008.
18. Lawrence S. Moss. Syllogistic logic with complements. In *Games, Norms and Reasons: Proceedings of the Second Indian Conference on Logic and its Applications*, page 19 pp. Springer Synthese Library Series, Mumbai, 2010.
19. Lawrence S. Moss. Notes on natural logics. unpublished ms., Indiana University, 2013.
20. Lawrence S. Moss. Syllogistic logic with cardinality comparisons. In Katalin Bimbo, editor, *J. Michael Dunn on Information Based Logics*, Outstanding Contributions to Logic. Springer-Verlag, to appear.
21. Reinhard Muskens. An analytic tableau system for natural logic. In Maria Aloni, Harald Bastiaanse, Tikitu de Jager, and Katrin Schulz, editors, *Logic, Language and Meaning*, volume 6042 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin Heidelberg, 2010.
22. Ian Pratt-Hartmann and Lawrence S. Moss. Logics for the relational syllogistic. *Review of Symbolic Logic*, 2(4):647–683, 2009.
23. John F. Sowa. Conceptual graphs: Online course in knowledge representation using conceptual graphs. `http://cg.huminf.aau.dk/index.html`. Accessed 1/18/2014.
24. Cameron Swords and Daniel Friedman. rKanren: Guided search in miniKanren. *Scheme and Functional Programming*, 2013.
25. Jan van Eijck. Natural logic for natural language. In *Logic, Language, and Computation*, volume 4363 of *LNAI*, pages 216–230. Springer-Verlag, 2007.
26. Adrian Walker. *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing*. Addison-Wesley, 1987.
27. Peter Yule and Buccleuch Place. A Prolog implementation of the method of Euler circles for syllogistic reasoning, 1996.