Universal Quantification and Implication in miniKanren

ENDE JIN, University of Toronto, Canada
GREGORY ROSENBLATT, University of Alabama at Birmingham, USA
MATTHEW MIGHT, University of Alabama at Birmingham, USA
LISA ZHANG, University of Toronto Mississauga, Canada

We present constructive universal quantification and implication in miniKanren using the goal constructors `(forall (v) domain goal)` and `(implies goal goal)`. Our implementation is capable of solving rudimentary logic problems while reasoning about equality, disequality, type constraints, and user-defined relations. The key idea behind handling a universal quantification `(forall (v) domain goal)` is to first run `(exists (v) (conj domain goal))` to find a state describing some v in the domain that satisfies the goal, then use a *relative complement* process to generate a goal `relcomp` describing the remaining space of v not covered by that state, and then recursively search `(forall (v) (conj relcomp domain) goal)`. The key idea behind handling an implication `(implies A B)` is to combine two strategies, depending on the form of `A`. One strategy follows the classical logic rule that $A \rightarrow B$ is equivalent to $\neg A \vee B$, and applies when a goal `A` can easily be negated. The other strategy uses syntactic pattern-matching, and applies when the antecedent goal `A` calls user-defined relations which match those appearing in the consequent goal `B`.

CCS Concepts: • **Software and its engineering** → **Constraint and logic languages**.

Additional Key Words and Phrases: miniKanren, logic programming, relational programming

1 INTRODUCTION

We present a method for implementing constructive universal quantification and implication in the constraint logic programming language miniKanren. In particular, we describe the goal constructors `(forall (v) domain goal)` and `(implies goal goal)`, extending miniKanren's existing set of goal constructors like `(fresh (v) goal)`, which is synonymous with `(exists (v) goal)`. These two new goal constructors allow miniKanren search to perform rudimentary theorem proving.

Here are some examples of the kind of goals that we can construct using `forall` and `implies`.

```
> (run 1 (a) (forall (v) () (== v a)))
'()                         ; no answers because no such "a" exists
> (run 1 () (forall (a) () (exists (v) () (== v a))))
'(())                       ; an answer exists and no extra information in the answer
> (run 1 (a b) (forall (v) () (disj* (== v a) (=/= v b))))
'((_.0 _.0))                ; can be made true if a == b
> (run 1 (b) (forall (a) () (disj* (not-symbolo a) (=/= a b))))
'((#f))                     ; can be made true if b == #f
> (run 1 (f) (implies (evalo f 1) (exists (a) (evalo f a))))
'((_.0))                    ; f can be anything
```

Authors' addresses: Ende Jin, University of Toronto, Toronto, ON, Canada, ende.jin@mail.utoronto.ca; Gregory Rosenblatt, University of Alabama at Birmingham, Birmingham, AL, USA, gregr@uab.edu; Matthew Might, University of Alabama at Birmingham, Birmingham, AL, USA, might@uab.edu; Lisa Zhang, University of Toronto Mississauga, Mississauga, ON, Canada, lczhang@cs.toronto.edu.

miniKanren.org/workshop/2021/8-ART2

Our motivation for introducing universal quantification (Section 3) and implication (Section 4) is to eventually be able to synthesize provably correct programs from universal properties. For example, we would like to be able to synthesize a provably correct sorting procedure from a universal specification. Along with an answer, we would also like to produce an independently checkable proof of the answer's correctness, as in Twelf and HOL [5]. As another example, we would like to prove theorems with computational content as is done with popular proof assistants such as ACL2, Coq and AGDA. Provably correct programs may be extracted from these proofs. Such a system could help programmers more easily write provably correct code.

To demonstrate progress towards these objectives, we include the following examples in Section 6:

- As a step towards synthesizing programs from universal properties, we work with a relational interpreter in Section 6.3.
- We reason about a sorting relation in Section 6.4.
- We discuss examples of reasoning in the presence of non-termination in Section 6.2 and Section 6.3.
- We discuss our system's relationship to intuitionistic logic, particularly regarding the Law of Excluded Middle in Section 6.6 and Section 6.2.

Our work is related to that of Ma et al. [4] and Moiseenko [7], both of which implement variations of universal quantification and implication. Both works are discussed in Section 5. Our work differs in that we aim for expressiveness in how these operations can be used (e.g. not being limited to stratified programs), and thus our approach trades off performance. We implement an experimental artifact[1] to illustrate our ideas.

2 BACKGROUND

We build on the first-order miniKanren implementation described in Rosenblatt et al. [8]. This implementation makes it easier to analyze and manipulate goals. For example, our implementation needs to track the scoping of logic variables, recognize when we've entered a `forall` scope, and syntactically match antecedent and consequent goals that call user-defined relations.

We begin with some definitions consistent with typical miniKanren implementations, and note where our interpretations differ.

Definition 2.1 (Literal Values $\mathcal{L}$)**.** We use $\mathcal{L}$ to denote the set of literal values in miniKanren. Our implementation of miniKanren supports numbers, symbols, booleans, strings, the empty list, and pairs. In other words, for a literal value `l` $\in \mathcal{L}$ we have `l = numbers .. | symbols .. | strings .. | #t | #f | '() | (cons l l)`.

Definition 2.2 (Logic Variables V and Terms T)**.** We use V to denote an infinite set of logic variables. Further, the set of terms T is defined such that for `t` $\in T$, we have either `t` $\in V$, `t` $\in \mathcal{L}$, or `t = (cons t1 t2)` with `t1, t2` $\in T$

Definition 2.3 (Substitution, Top, Bottom)**.** A substitution σ is a function that maps logic variables to terms. We represent substitutions using the notation $\sigma = [v_1 \mapsto t_1, ..., v_n \mapsto t_n]$. Additionally, we use $\top$ to denote the empty substitution, and $\bot$ to represent failure.

Definition 2.4 (State STATE)**.** A state consists of a substitution, a set of type constraints, and a set of disequality disjunctions. Our implementation of miniKanren supports the type constraints `numbero` , `symbolo` , `stringo` , `pairo` and their negations (e.g. `not-numbero`).

Definition 2.5 (Atomic Goal, Goals $\mathcal{G}$)**.** We define an *atomic goal* to be an equality, disequality, or type constraint. Moreover, we exclude `cons` from appearing in an atomic type constraint.

We define $\mathcal{G}$ to be the set of all goals in miniKanren, with $g \in \mathcal{G}$ if g is atomic, g is an equality or disequality constraint that uses `cons` , or `g = (conj g g) | (disj g g) | (exists (x) g) | (Top) | (Bottom)`. Here, `(Top)` is a goal that can be trivially satisfied (i.e. running this goal makes no changes to the state). We will sometimes represent this goal using the same symbol $\top$ used for empty substitutions. Moreover, `(Bottom)` cannot be satisfied (i.e. running this goal will result in no answers). We will sometimes represent this goal using the symbol $\bot$.

[1] https://github.com/DKXXXL/fo-mKanren-to-LF

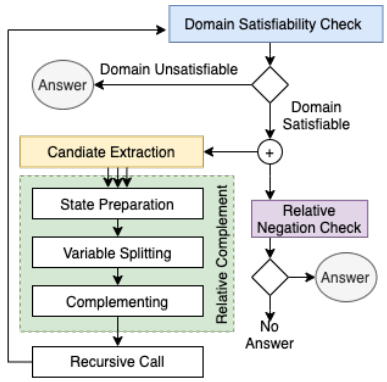

Fig. 1. Steps in the universal quantification search algorithm.

We choose the goal constructor name `exists` rather than the more typical `fresh` to emphasize the logical relationship with `forall`. In the following section, $\mathcal{G}$ may include new goals like universal quantification, implication and calls to user-defined relations according to the context.

Proposition 2.1 (Goal representation of a State)**.** A state can be rewritten as a goal that represents the same constraints as the state: when we run the goal, we should get the equivalent state back.

We use Proposition 2.1 throughout the paper, both implicitly to visualize states, and explicitly when we manipulate these states.

3 UNIVERSAL QUANTIFICATION

This section introduces the goal constructor `(forall (v) domain body)`, where `v` is the universally quantified logic variable, `domain` is a goal that restricts the portion of $v \in \mathcal{L}$ that we consider, and `body` is a goal that we wish for all choices of v in `domain` to satisfy. In the case where the `domain` is all of $\mathcal{L}$, we use the syntax `(forall (v) () body)`. We otherwise restrict `domain` goals to conjunctions and disjunctions of atomic goals.

Figure 1 shows the steps involved in satisfying a universal quantification goal:

(1) **Domain Satisfiability Check** First, we attempt to satisfy the goal `domain`. If `domain` is unsatisfiable, the universal quantification succeeds vacuously. We describe the base case check in Section 3.1.
(2) **Candidate Extraction** Next, we attempt to satisfy the goal `(exists (v) (conj domain body))`, yielding a stream of states where each expresses how `body` can succeed for some subset of $v \in S \subset \mathcal{L}$. If this stream is empty, the universally quantified goal cannot be satisfied unless there is a way to falsify the domain. We describe the candidate extraction in Section 3.2.
(3) **Relative Complement** Suppose candidate extraction succeeded with a state $S \subset \mathcal{L}$. For the universally quantified version of the goal to succeed, we need to consider whether the remaining literals $v \in \mathcal{L} \backslash S$ satisfy the goal `body`. We derive a goal `relcomp` that describes this remaining search space. With the complemented goal `relcomp`, we can continue the search recursively with the goal `(forall (v) (conj relcomp domain) body))`. We describe the relative complement algorithm in Section 3.3.
(4) **Domain Relative Negation Check** Rather than trying to fulfill the `body` goal along the `domain`, an alternative approach is to falsify the *domain*. This strategy is described in Section 3.4.

The rest of this section describes each of these four steps. For consistency, we use v to denote the universally quantified logic variable, symbols a, b, c to denote logic variables *external* to the scope of the `forall` and symbols x, y, z to denote logic variables *internal* to the scope of the `forall`.[2] A proof of this universal quantification handling algorithm is in the Appendix A.1.

[2]For example, consider the goal `(exists (a b c) (forall (v) () (exists (x y z) etc ...)))`

Before presenting each of the four steps, we present two example `forall` goals to demonstrate the intuition behind these steps.

Example 3.1. Consider the goal `(forall (v) () (disj (== v 1) (=/= v 1) (== v 2)))`. We use this goal to illustrate steps 1-3 of the algorithm, and omit the relative negation check for now (since it produces no answers). Here's what happens when we run the goal:

(1) Domain Satisfiability Check: There is no explicit domain goal to check. The domain of v is $\mathcal{L}$, which clearly succeeds.
(2) Candidate Extraction: We check if `(exists (v) (disj (== v 1) (=/= v 1) (== v 2)))` succeeds. This goal succeeds, and we obtain a state with the substitution $[v \rightarrow 1]$. It remains to check the remaining possible values of v, namely $\mathcal{L} \setminus 1$
(3) Relative Complement: The returned state can be represented by the goal `(== v 1)`. The relative complement of this goal (capturing the "remaining" $v \in \mathcal{L}$) is the goal `(=/= v 1)`. We continue the search with the reduced domain: `(forall v (=/= v 1) (disj (== v 1) (=/= v 1) (== v 2)))`

This recursive call to `forall` continues with the following steps:

(1) Domain Satisfiability Check: The goal `(=/= v 1)` succeeds.
(2) Candidate Extraction: We check if `(exists (v) (conj (=/= v 1) (disj (== v 1) (=/= v 1) (== v 2))))` succeeds. This goal should succeed, and we should obtain a state with the disequality constraint $v \neq 1$.
(3) Relative Complement: This state can be represented by the goal `(=/= v 1)`, whose relative complement is `(== v 1)`. We continue the search with the reduced domain: `(forall v (conj (== v 1) (=/= v 1)) (disj (== v 1) (=/= v 1) (== v 2)))`

Here, we see that the domain goal `(conj (== v 1) (=/= v 1))` is not satisfiable, meaning we have reached our base case and obtain an answer.

Example 3.2. To demonstrate the way universal quantification interacts with search, consider the following goal with a universal quantifier inside an existential quantifier: `(exists (a b) (forall (v) () (disj (== v a) (=/= v b))))`. Readers well-versed in logic will see that this goal is satisfiable with the state `(== a b)`. We illustrate how our universal quantification algorithm arrives at such an answer. For readability, we only show the two search paths that produce the answer. For both paths, the following steps occur first:

(1) When entering the `exists` goal, miniKanren creates two fresh logic variables `a` and `b`.
(2) When entering the `forall` goal, we check that the domain is clearly satisfiable.
(3) We run the candidate extraction goal `(exists (v) (disj (== v a) (=/= v b)))`.

One way to produce the answer is to use the first state from the candidate extraction, which is rewritten to the goal `(== v a)`.

(4) The relative complement of `(== v a)` is `(=/= v a)`.
(5) We run the goal `(forall (v) (=/= v a) (disj (== v a) (=/= v b))`.
(6) The domain goal `(=/= v a)` is satisfiable.
(7) We run a second candidate extraction `(exists (v) (conj (=/= v a) (disj (== v a) (=/= v b))))`.
(8) The first state we obtain is `(conj (=/= v a) (=/= v b))`.
(9) Its relative complement is `(disj (== v a) (== v b))`.
(10) We run the goal `(forall (v) (conj (=/= v a) (disj (== v a) (== v b))) (disj (== v a) (=/= v b)))`.
(11) The domain is equivalent to `(conj (=/= v a) (=/= a b))`, and relative negation will falsify it to extract an answer. One way to make this goal fail is to set `(== a b)`.

A different way of obtaining the same answer is to use the second state we obtain from the initial candidate extraction, rewritten to the goal `(=/= v b)`.

(4) The relative complement of `(=/= v b)` is `(== v b)`.
(5) We then run the goal `(forall (v) (== v b) (disj (== v a) (=/= v b))`.
(6) The domain goal `(== v b)` is satisfiable.

(7) We run a second candidate extraction `(exists (v) (conj (== v b) (disj (== v a) (=/= v b))))`.
(8) The first state we obtain is `(conj (== v b) (== v a))`, which is rewritten to `(conj (== v b) (== v a) (== a b))` to expose the implied constraint on a and b.
(9) The relative complement is `(conj (== a b) (disj (=/= v a) (=/= v b)))`.
(10) We run the goal `(forall (v) (conj (=/= v a) (disj (== v a) (== v b)) (conj (== a b) (disj (=/= v a) (=/= v b)))) (disj (== v a) (=/= v b)))`.
(11) **The domain goal is not satisfiable, producing another answer that is also `(== a b)`.** This happens to be the same answer that we obtained earlier, but we used a different approach to arrive at this answer.

3.1 Base Case

In this step, we check if the `domain` goal is satisfiable by running the goal `(exists (v) domain)`. If `domain` is not satisfiable, then the goal `(forall (v) domain body)` succeeds vacuously and we get an answer.

If `domain` succeeds, then we need to continue to the candidate extraction step. Since `domain` will be used again in the candidate extraction step, as an optimization, we prune disjuncts within `domain` that are guaranteed to fail. For example, if `domain` is the goal `(disj (=/= v v) (== v 1))`, then in the course of checking its satisfiability, we will learn that `(=/= v v)` is not satisfiable, and will simplify `domain` to the goal `(== v 1)` before proceeding to the next step.

3.2 Candidate Extraction

In this step, we try to satisfy the goal `(exists (v) (conj domain body))`, yielding a stream of states, and we proceed with the next step for *each* of these states. To satisfy a precondition in the next step, we further split each state into one or more *disjunction-free* states.

Definition 3.1 (Disjunction-Free States)**.** A state is disjunction-free if it can be expressed as a conjunction of equality, type, and disequality constraints.

Recall that a state consists of a substitution (a conjunction of equality constraints), a conjunction of type constraints, and a conjunction of disjunctions of disequality constraints. Each disjunction of disequality constraints needs to be split.

Example 3.3. Consider the state denoted by the goal `(conj (== v 1) (== x 1) (numbero a) (disj (=/= v a) (=/= v b)))`. This state is not disjunction-free, but can be split into two disjunction-free states:

```
(conj (== v 1) (== x 1) (numbero a) (=/= v a))
(conj (== v 1) (== x 1) (numbero a) (=/= v b))
```

3.3 Relative Complement

In this step, we begin with a disjunction-free state that satisfies `(exists (v) (conj domain body))`, and wish to obtain a *relative complement* of the state, namely a goal `relcom` that expresses the "remaining" part of $v \in \mathcal{L}$ that isn't already captured by this state. This idea is summarized in Proposition 3.1.

Proposition 3.1 (Relative Complement and the semantics of `forall`)**.** Given a state from `(exists (v) (conj domain goal))` expressed as the goal `state0`, and its *relative complement* `relcomp` (to be defined in Definition 3.2), we have that any state that is an answer to the goal: `(conj state0 (forall (v) (conj relcomp domain) body))` should also be an answer to the goal `(forall (v) domain body)`.

Intuitively, `state0` assures that for a portion of $v \in \mathcal{L}$, the goal `body` is true. The recursive call to `forall` checks the remaining subset of $\mathcal{L}$ not covered by `state0`.

Definition 3.2 (Relative Complement)**.** A relative complement $R(\mathrm{g}, v, \{a, b, ...\}, \{x, y, ...\})$ of a goal `g` with respect to the universally quantified logic variable v, external logic variables $\{a, b, \dots\}$, and internal logic variables $\{x, y, \dots\}$, is a goal `relcomp` with the following properties:

- **Coverage and disjointness:** For any choice of $v_0 \in \mathcal{L}$, the goal `(conj g (== v` v_0`))` succeeds if and only if the goal `(conj relcomp (== v` v_0`))` fails.

- **External consistency:** For any choice of external variables, i.e. $a_0 \in \mathcal{L}, b_0 \in \mathcal{L}, \ldots$, if both `g` and `relcomp` individually succeed, then the goal `(conj g (== a` a_0`) (== b` b_0`) ...)` succeeds if and only if the goal `(conj relcomp (== a` a_0`) (== b` b_0`) ...)` succeeds.

Notice that since the internal variables can be different for each choice of v, we don't need to restrict the choice of those variables.

Although the relative complements of a goal are not syntactically unique[3], we provide a construction of a goal that we refer to as *the* relative complement. We describe the construction for *atomic* goals, before working up to any goals that come from a disjunction-free state. The final algorithm for the construction is described at the end of this section.

Definition 3.3 (Complement)**.** A complement of a goal `g` is a goal $\overline{\mathtt{g}}$ such that `(disj g` $\overline{\mathtt{g}}$`)` succeeds and `(conj g` $\overline{\mathtt{g}}$`)` fails for any choice of the logic variables in `g`. In particular, for an atomic goal `g`, its complement can be obtained by interchanging equality and disequality constraints, and type and not-type constraints.

Proposition 3.2 (Relative Complement of an atomic goal)**.** A relative complement $R(\mathtt{g}, v, \{a, b, ...\}, \{x, y, ...\})$ of an atomic goal `g` $\in \mathcal{A}$ can be obtained as follows:

- If `g` references logic variables $\{x, y, ...\}$ scoped inside the `forall`, then $R(\mathtt{g}, v, \{a, b, ...\}, \{x, y, ...\}) = \top$.
- If `g` references v (but not $x, y, ...$), then $R(\mathtt{g}, v, \{a, b, ...\}, \{x, y, ...\}) = \overline{g}$.
- If `g` only references logic variables $\{a, b, \ldots\}$ scoped outside of the `forall`, then $R(\mathtt{g}, v, \{a, b, ...\}, \{x, y, ...\}) = g$.

Example 3.4. We use Proposition 3.2 to construct the relative complement of the following goals with respect to v, $\{a, b, \ldots\}$ and $\{x, y, \ldots\}$:

- The relative complement of the goal `(== v a)` is `(=/= v a)`.
- The relative complement of the goal `(conj (== b 2) (== v a))` is `(conj (== b 2) (=/= v a)`.
- The relative complement of the goal `(pairo v)` is `(not-pairo v)`.
- The relative complement of the goal `(== a 1)` is $\bot$, because v is unconstrained in `(== a 1)`.

Next, we extend the relative complement derivation to a *conjunction* of atomic goals satisfying certain conditions:

Definition 3.4 (Conjunction-Only Goal $C \subset \mathcal{G}$)**.** A goal $\mathtt{g} \in C$ is **conjunction-only** if it is a conjunction of atomic goals. Recall that an atomic goal disallows the use of `cons`.

Proposition 3.3 (Relative Complement of a Conjunction-Only Goal C)**.** Suppose $\mathtt{g} \in C$, with

$$\mathtt{g} = \bigwedge_i \alpha_i \wedge \bigwedge_j \mu_j \wedge \bigwedge_k \chi_k$$

where each $\alpha_i, \mu_j, \chi_k \in \mathcal{A}$ is atomic, the α_is reference only the variables $\{a, b...\}$ outside the scope of the `forall`, and the μ_js reference the universally quantified variable v (and potentially $\{a, b...\}$), and the χ_ks reference variables $\{x, y...\}$ inside the scope of the `forall`.

Moreover, all implied constraints in `g` on the universally quantified variable have been expressed in the μ_is, and likewise all implied constraints in `g` on the external variables have been expressed in the α_is. (For example, if `g` contains the atomic conjuncts `(= a v)` and `(= b v)`, then it also contains `(= a b)`.)

Then the relative complement of g is:

$$\bigwedge_i \alpha_i \wedge \bigvee_j \overline{\mu_j}$$

Example 3.5 (Relative Complement of Conjunction-Only Goals)**.** Consider the relative complement of the following goals with respect to v, $\{a, b, \ldots\}$ and $\{x, y, \ldots\}$.

[3]they are only unique up-to logical equivalence, but can have multiple syntactical form. `(== a b)` has `(=/= a b)` as one relative complement, but also `(conj (=/= a b) (== a a)` as another

- The relative complement of the goal `(conj (== a 1) (=/= v b) (symbolo v) (== a x)` is `(conj (== a 1) (disj (== v b) (not-symbolo v)))`.
- The goal `(=/= v (cons a x))` is not conjunction-only since it uses `cons`, so Proposition 3.3 does not apply. The relative complement of this goal is *not* `(== v (cons a x))`. Instead, the relative complement should behave like the following `(disj (not-pairo v) (conj (pairo v) (=/= (car v) a)))` —although we can't use `(car v)` to destruct a logic variable directly.
- The goal `(conj (== v a) (== v b))` has some implied constraints between the external variables `a` and `b`, so Proposition 3.3 does not apply. Indeed, the relative complement of this goal is *not* `(disj (=/= v a) (=/= v b))`. It should instead capture the fact that `(== a b)`.

Unfortunately, the states we get from candidate extraction do not directly translate to goals that satisfy the conditions in Proposition 3.3. In particular, a disjunction-free state might nevertheless have `cons` in its equality and disequality constraints. To solve this issue, we extend the notion of logic variables to include field projections of logic variables, such as `(car v)`, `(cdr v)`, and arbitrary nestings such as `(car (cdr v))`. We call these *projected logic variables*.

Definition 3.5 (Projected Logic Variables V^P)**.** We extend the set of logic variables V with its *projection* V^P, where $V \subset V^P$, and if $v \in V^P$, then `(car v)`, `(cdr v)` $\in V^P$. To avoid confusion, we use the notations `x.car`, `x.cdr` to indicate `(car x)`, `(cdr x)` respectively. We extend the definition of substitution to allow projected logic variables in the domain.

With these projected variables, we can rewrite states so that substitutions and disequality constraints never involve `cons`.

Definition 3.6 (Field projection form)**.** A goal or state is in *Field Projection Form* if there is no mention of `cons` anywhere in (the encoding of) the goal or state. Instead, we explicitly reference the components of any `cons`. In our implementation, `cons` never appears in type constraints.

Example 3.6 (Field Projection)**.** Consider the goal `(== v (cons a (cons b c)))`. We can rewrite this goal in field-projected form: `(conj (== v.car a) (== v.cdr.car b) (== v.cdr.cdr c)`.

Moreover, consider the goal `(=/= v (cons a b))`. We can rewrite this goal in field-projected form: `(disj (not-pairo v) (=/= v.car a) (=/= v.cdr b))`.

Proposition 3.4 (Extending Proposition 3.3 to Field-Projected Goals)**.** We can apply the construction in Proposition 3.3 to find the relative complement of a conjunction-only, field-projected goal `g` if for any field-projected logic variable k (i.e. the goal refers `k.car` or `k.cdr`) the type constraint `(pairo k)` appears in `g`.

Example 3.7 (Field Projected Relative Complement)**.** Consider the relative complement of the following goals with respect to v, $\{a, b, \dots\}$ and $\{x, y, \dots\}$: `(conj (=/= v.car a) (=/= v.cdr b) (pairo v))`. The relative complement of this goal is `(disj (== v.car a) (== v.cdr b) (not-pairo v))`. The reason why the type constraint is required is because our system will translate a constraint like `(== v.car a)` back into the following usual miniKanren goal: `(exists (x) (== v (cons a x)))`.

We summarize our relative complement implementation in Algorithm 3.1.

Algorithm 3.1 (Relative Complement of a Disjunction-Free State)**.** We apply these steps to compute the relative complement of a disjunction-free state:

(1) Apply field projection, and add any missing `pairo` type constraints.
(2) Make any implied constraints on the constraints on the universally quantified and external logic variables explicit.
(3) Remove internal logic variables from the state, and separate information related to `v` versus those unrelated to `v` (to be able to apply Proposition 3.3).
(4) Split any newly-introduced disjunctions as necessary.
(5) Apply Proposition 3.2 on the atomic goals that mention `v` to obtain the desired goal.

A proof of this algorithm is attached in the Appendix A.2.

With the relative complement `relcomp`, we can run the goal `(forall (v) (conj relcomp domain) body)`. Notice that when running this goal, it is important to use the *state* containing any constraints related to the external variables. If `(conj relcomp domain)` fails during the recursive step, we return the state with these constraints on the external variables.

3.4 Domain Relative Negation

This section presents an alternative strategy to find answers to the goal `(forall (v) domain body)`. To find these answers, we consider the external logic variables that are mentioned in the goal `domain`, and find assignments for those variables that cause `domain` to fail. We find such assignments by running the *relative negation* of the goal `domain`, obtained by flipping the constraints that `domain` places on the external variables. The approach is summarized in Proposition 3.5.

Proposition 3.5 (Domain Relative Negation and the semantics of `forall`)**.** Suppose the *relative negation* of the goal `domain` is the goal `negdomain`. Then any solution to the goal:

```
(exists (a b ...) negdomain)
```

should also be a solution for the goal `(exists (a b ...) (forall (v) domain body))`.

Definition 3.7 (Relative Negation)**.** A relative negation $N(\mathtt{g}, \{a, b, \dots\})$ of a goal `g` with respect to the external logic variables $\{a, b, \dots\}$ is a goal ¬g with the property that for any choice of external variables $a_0 \in \mathcal{L}, b_0 \in \mathcal{L}, \dots$, the goal `(conj g (== a` a_0`) (== b` b_0`) ...)` succeeds if and only if `(conj ¬g (== a` a_0`) (== b` b_0`) ...)` fails.

To illustrate how domain relative negation can find answers that the previous sections miss, consider this example:

Example 3.8. The goal `(exists (a b) (forall (v) (conj (== v a) (=/= v b)) (=/= v v)))` has a domain goal `(conj (== v a) (=/= v b))` that is satisfiable, and will pass the domain satisfiability check. Since the body goal `(=/= v v)` is clearly not satisfiable, candidate extraction fails. However, the relative negation of the domain goal is `(== a b)`. With this constraint, the `forall` goal can succeed vacuously.

Proposition 3.6 (Relative Negation Construction)**.** Consider a goal `g`, which is a disjunction of field-projected, conjunction-only goals:

$$\mathtt{g} = \bigvee_n \left(\bigwedge_i \alpha_{n,i} \wedge \bigwedge_j \mu_{n,j} \right)$$

where each $\alpha_{n,i}, \mu_{n,j} \in \mathcal{A}$ is atomic (and possibly field-projected), and the α_is reference only the variables $\{a, b...\}$ outside the scope of the `forall`. Moreover, all implied constraints in `g` on the external variables have been made explicit in the α_is.

Then the relative negation of g is:

$$\bigwedge_n \bigvee_i \overline{\alpha_{n,i}}$$

Proposition 3.6 provides an approach to negate goals that, when field-projected, only contain conjunctions, disjunctions, and atomic goals. Any such goal can be written in disjunctive normal form.

A proof of this proposition is attached in the appendix.

4 CONSTRUCTIVE IMPLICATION

This section introduces the goal constructor `(implies antecedent consequent)` where `antecedent` and `consequent` are goals. For antecedent goals that we can easily negate (like those in Proposition 3.6), we can rewrite the implication goal to `(disj neg-antecedent consequent)`. We only negate goals that are primitive constraints. This use of negation is sufficient for handling the primitive constraints because they are decidable. Unfortunately, other goals, such as calls to user-defined relations, are not easily negatable. Running these goals might not even terminate. To support a general treatment of implication, we supplement

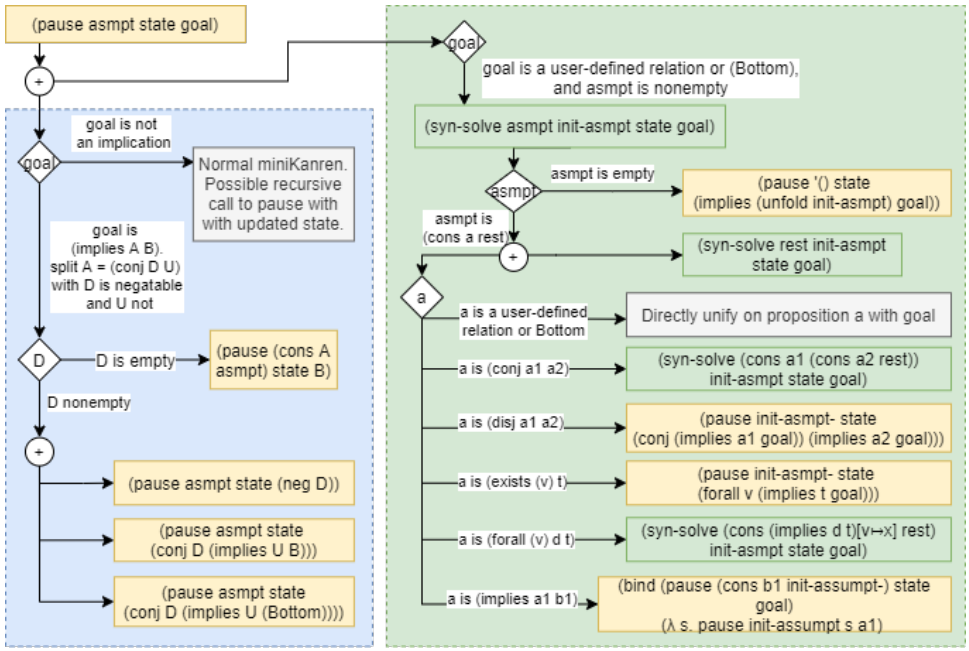

Fig. 2. Computation flow for miniKanren with constructive implication, with semantic solving on left (blue) and syntactic solving on right (green). The symbol ⊕ denotes mplus stream addition and diamonds represent cond dispatch.

miniKanren's `pause` function with an extra argument referencing the list of accumulated assumptions. Recall that `pause` also takes the current `state` and a target `goal`, and is the heart of the miniKanren search algorithm. Our implementation of `pause` interleaves two search strategies: *semantic search* (Section 4.1) and *syntactic search* Section 4.2. A summary of the process flow of the `pause` function is shown in Figure 2.

Example 4.1 (Semantic vs Syntactic Solving)**.** As an example of semantic solving, the goal `(implies (== a 1) (== a b))` can be rewritten to the goal `(disj (=/= a 1) (== a b))`.

As an example of a goal that cannot be solved semantically, suppose we have a user-defined relation `dead-loop` that does not terminate. The goal `(exists (a) (implies (dead-loop 1) (dead-loop a))` should succeed with the substitution $[a \mapsto 1]$. This goal cannot be solved using semantic solving, since the antecedent goal is not negatable. However, syntactic solving will apply pattern matching to the antecedent and consequent calls to `dead-loop`, succeeding with the state `(== a 1)`.

4.1 Semantic Solving

Semantic solving involves using miniKanren's existing search strategy, and is triggered for all goals when calling `(pause asmpt state goal)`. If `goal` is an implication `(implies antecedent consequent)`, we want to apply the logic rule that $A \rightarrow B$ is equivalent to $\neg A \vee B$ as often as possible. However, we cannot negate goals that call user-defined relations. To be more specific, in order to negate a goal, it must have the property defined in Definition 4.1.

Definition 4.1 (Negatable). A goal `g` is negatable with the negation ¬g if one of the following is true:

- `g` is an equaliy, disequality, type, or not-type goal. To obtain ¬g, swap equality and disequality relations, and type and not-type relations.
- `g` is a conjunction of negatable goals. Here, `¬(conj g1 g2) = (disj ¬g1 ¬g2)`
- `g` is a disjunction of negatable goals. Here, `¬(disj g1 g2) = (conj ¬g1 ¬g2)`

- `g` is an implication, of which the components are negatable. Here, `¬(implies g1 g2) = (conj g1 ¬g2)`
- `g` is an existential quantifier with a negatable body. Here, `¬(exists (x) g) = (forall (x) ¬g)`
- `g` is a universal quantifier with a negatable body. Here, `¬(forall (x) domain g) = (exists (x) (conj ¬domain ¬g))`

To use the existing miniKanren search as much as possible, we split `antecedent` into a conjunction of a negatable and unnegatable components `(conj negatable unnegatable)`. Our implication goal can then be rewritten as `(implies (conj negatable unnegatable) consequent)`.

Suppose first that the `negatable` goal is nonempty. In this case, we apply the following proposition to separately handle `negatable` and `unnegatable`.

Proposition 4.1 (Semantic Solving of Implication)**.** Suppose `negatable` is nonempty. Then there are three possible disjoint ways to satisfy the goal `(implies (conj negatable unnegatable) consequent)`:

(1) If `negatable` can be made to fail. To find a state that makes `negatable` fail, run the goal `¬negatable`.
(2) If it is possible to satisfy `negatable`, while making `unnegatable` fail. To find such a state, first satisfy `negatable`, then run `(implies unnegatable (Bottom))`. In other words, run the goal `(conj negatable (implies unnegatable (Bottom)))`. The special consequent `(Bottom)` signifies failure, and is used by our system to trigger syntactic solving.
(3) If it's possible to satisfy both `negatable` and `(implies unnegatable consequent)`. In other words, run the goal `(conj negatable (implies unnegatable consequent))`.

Notice that in the latter two queries we use the fact that in miniKanren conjunctions, the first conjunct is always processed first. When processing the second conjunct (e.g. when `pause` is called with the second conjunct goal), the information from the first conjunct is stored in the *state*.

Example 4.2. Consider the antecedent goal `(conj (== x 1) (foo y) (disj (== y 2) (bar x y)))` where `foo` and `bar` are user-defined relations. The negatable part of this goal is the goal `(== x 1)` and the non-negatable part is the goal `(conj (foo y) (disj (== y 2) (bar x y)))`.

Proposition 4.1 does not apply in the case that there is no negatable component to the antecedent in `(implies antecedent consequent)`. If `antecedent` is not negatable at all, we need to run the goal `consequent`. However, when we make the recursive call to `pause`, we add `antecedent` to the list of assumptions. The assumptions are not used by the semantic solver, but the syntactic solver will make use of this parameter.

Example 4.3. Consider the goal `(implies (disj (== y 2) (bar x y)) (bar y y))` where `bar` is a user-defined relation. Since the antecedent is unnegatable, we run this goal by running the consequent goal `(bar y y)` with an additional assumption `(disj (== y 2) (bar x y))`, triggering the syntactic solver.

4.2 Syntactic Solving

The syntactic solver is triggered when calling `(pause asmpt state goal)` where `goal` is a user-defined goal or the special goal `(Bottom)`, and `asmpt` is nonempty. The syntactic solver iterates over the list of assumptions `asmpt`, and checks whether the assumptions can be used to satisfy `goal`. For a simple assumption, this check is done using an extended unification algorithm that uses pattern matching on the goals and assumptions (Proposition 4.2). For compound assumptions like conjunctions, disjunctions and other goals, we break the assumption into parts and recursively handle each part (Proposition 4.3). When the list of assumptions becomes empty, we try to *unfold* the user-defined relations in the initial assumption (see Proposition 4.4).

The syntactic solver is recursive with the function signature `(syn-solver asmpt init-asmpt state goal)`. When entering `syn-solver` from `pause`, the parameter `init-asmpt` is set to `asmpt`. We use the variable name `a` to denote the first element of `asmpt`. We have two choices with respect to `a`: either we actively apply the assumption goal `a` to make `goal` succeed, or we don't apply `a` and use the rest of `asmpt`. These two choices are reflected by the $\oplus$ symbol on the right side of Figure 2.

Actively using `a` to run `goal` is the core of the syntactic solver. We begin with the simplest case, where `a` is a user-defined relation or `Bottom`.[4]

Proposition 4.2 (Unification with an assumption goal)**.** A goal `(Bottom)` succeeds if and only if the assumption `a` is `(Bottom)`. A goal which calls the user-defined relation `(relation x1 x2 ...)` succeeds if and only if the assumption `a` is `(relation y1 y2 ...)` where `x1` unifies with `y1`, and `x2` unifies with `y2`, and etc.

The above proposition serves as a base case for using an assumption to solve a goal. When an assumption `a` is not a call to a user-defined relation, we need to split `a` into parts.

Proposition 4.3 (Compound assumptions)**.** Suppose the assumption goal `a` is not a user-defined relation or `(Bottom)`. Then the implication can be solved as follows.

- If `a` is the goal `(conj a1 a2)`, then we can treat `a1` and `a2` as two separate assumptions in the list `asmpt`. Proceed as if `a1` is the first element of `asmpt`, and `a2` is in the rest of `asmpt`.
- If `a` is the goal `(disj a1 a2)`, then solving the `goal` is equivalent to solving the alternative goal `(conj (implies a1 goal) (implies a2 goal))` *with reduced assumptions* `init-asmpt-`.
- If `a` is the goal `(exists (v) t)`, then solving the `goal` is equivalent to solving the alternative goal `(forall v (implies t goal))` *with reduced assumptions* `init-asmpt-`.
- If `a` is the goal `(forall (v) d t)`, then we can treat the instantiated `a` (i.e. `(implies d t)` $[v \mapsto x]$) as the replaced assumption and proceed. Here, the logic variable `v` is substituted with a fresh logic variable `x`.
- If `a` is the goal `(implies a1 b1)`, then in order to use this assumption we need to satisfy `a1` while also satisfying `goal` with assumption `(cons b1 init-asmpt-)` where `init-asmpt-` is *the reduced assumption.*

Here, the reduced assumption set `init-asmpt-` is the result of removing the current assumption. Removing the current assumption is necessary to prevent infinite looping when solving the equivalent new goal.

As we iterate over these assumptions in the list of goals `asmpt`, we recursively call the function `syn-solve`, stripping off the elements of `asmpt` in each iteration. However, we do not change the parameter `init-asmpt`, leaving us with access to the list of initial assumptions when `asmpt` becomes empty.

When `asmpt` becomes empty, our search is not over. The user-defined relations in `asmpt` might still have information that will help us in solving `goal`. The following example illustrates a case where *unfolding* the definition of a relation is useful.

Example 4.4. Suppose we have the following relations:

```
(define-relation (foo x) (foo (cons 1 x)))
(define-relation (bar x) (foo x))
```

Then we would expect the goal `(exists (x) (implies (foo 2) (bar (cons x 2))))` to succeed with $[x \mapsto 1]$. The reason is that `(foo 2)` implies `(foo (cons 1 2))`, which then implies `(bar (cons 1 2))`.

Unfolding a relation in this way is valid under a closed-world interpretation of a relation definition, treating it as an *equivalence* between a relation call and its body.

That is, a goal that calls the relation succeeds if *and only if* the body of the relation succeeds. This is different from a typical open-world interpretation of a user-defined relation as a definite Horn clause[1], where the body implies the relation call, but the relation call doesn't have to imply the body.

To be concrete, in Example 4.4, not only does `(foo (cons 1 x))` imply `(foo x)`, but also `(foo x)` implies `(foo (cons 1 x))`.

Relation definitions can also be viewed as extending the logic with both a new introduction rule[2] and a corresponding elimination rule[3].

We believe this interpretation of user-defined relations is a more convenient default for reasoning about the relations we're interested in, such as `evalo`, which are intended to be complete definitions.

[4]Recall that the built-in equality, disequality, and type constraints are handled by the semantic solver.

Definition 4.1 (Unfolding)**.** The unfolding of a goal `g` is a goal where every sub-goal that calls a user-defined relation is replaced with its body.

Proposition 4.4 (Unfolding assumptions and goal)**.** Any state that is a solution of the goal:

`(implies (unfold asmpt) goal)`

should also be a solution to the goal `(implies asmpt goal)`.

We use Proposition 4.4 when `asmpt` is empty, by unfolding `init-asmpt`, and calling `pause` with the resulting goal `(implies (unfold init-asmpt) goal)`. We call `pause` and not `syn-solve` since the goal `(unfold init-asmpt)` can have a negatable component, even when `init-asmpt` does not.

Example 4.5. To demonstrate the way the solver works, we trace through what happens when running the goal from Example 4.4: `(exists (x) (implies (foo 2) (bar (cons x 2))))`. We only describe the search path that produces an answer:

(1) The entry point for solving the goal is a call to `(pause '() st (implies (foo 2) (bar (cons x 2))))`.
(2) Since the antecedent has no negatable component, we add the antecedent to the list of assumptions and run `(pause (list (foo 2)) st (bar (cons x 2)))`.
(3) Both the syntactic and semantic solver will trigger. The syntactic solver fails. The semantic solver will use normal miniKanren to expand the current goal, and calls `(pause (list (foo 2)) st (foo (cons x 2)))`.
(4) Both the syntactic and semantic solver will trigger. In the syntactic solver, since the first assumption `(foo 2)` is a relation, we try to use it to match `(foo (cons x 2))`, which fails.
(5) Since the rest of the list of assumptions is empty, we unfold the assumptions and run the goal `(implies (foo (cons 1 2)) (foo (cons x 2)))`.
(6) Again, the antecedent to the list of assumptions has no negatable part. Again, the syntactic solver will trigger. Again, the first assumption `(foo (cons 1 2))` is a user-defined relation. This time, the assumption uses the same relation as the goal `(foo (cons x 2))`, and the argument can be unified with $[x \mapsto 1]$.

5 RELATED WORK

The miniKanren version at http://github.com/webyrd/miniKanren-with-symbolic-constraints implements a limited form of universal quantification in miniKanren, introducing a goal constructor called `eigen` which is analogous to `fresh`. In this implementation, variables introduced by `eigen` are compatible with equality constraints but do not support type or disequality constraints.

Ma et al. [4] introduces λKanren, which is inspired by λProlog [6]. The work implements universal quantification by introducing the goal constructor called `all`. The only constraints implemented in λKanren are equality constraints, making it difficult to express many of the queries we're interested in. λKanren also implements a limited form of implication, introducing the goal constructor `assume-rel`. With this implementation of implication, the only goals supported in antecedent positions are relation calls and equality constraints. No unfolding of antecedent goals is performed, which is consistent with an open-world interpretation of relations defined as definite Horn clauses.

Moiseenko [7] implements stratifiable, constructive negation in miniKanren, and builds universal quantification and implication operators in terms of negation. In contrast, our approach implements universal quantification and implication as the primitives, and we express negation in terms of implication. Our approach has no stratification restriction on the use of negation, allowing more programs to be expressed. However, the constructive negation approach [7] has a simpler implementation, and we believe it will generally run faster on queries where both approaches provide answers. We use the test suite from Moiseenko [7] in Section 6.1.

6 EVALUATION

In this section, we demonstrate the capabilities of our universal quantification and constructive implication. As is miniKanren tradition, search can be slow. Still, we can express and run some interesting queries. We also discuss their performance characteristics.

Table 1. Basic Quantifier Examples from Moiseenko [7]. We use Greek letters $\alpha, \beta, \ldots$ to represent query variables, and always query for 1 answer.

Query	Result	Time (ms)
$\forall x.(x = \alpha)$	()	3
$\forall x.\exists y.(x = y)$	$(((\_.0).(\top)))$	1
$\forall x.\exists y.((x = y) \wedge (y = \alpha))$	()	3
$\forall x.(\alpha = (1, x))$	()	22
$\forall x.\exists y.(y = (1, x))$	$(((\_.0).(\top)))$	1
$\forall x.\exists y.(x = (1, y))$	()	4
$\forall x.(x \neq \alpha)$	()	3
$\forall x.\exists y.(x \neq y)$	$(((\_.0).(\top)))$	1
$\forall x.\exists y.((x \neq y) \wedge (y = \alpha))$	()	3
$\forall x.(\alpha \neq (1, x))$	$(((\_.0).('\_.0 \in$ `'(number? symbol? string?))))`	1
$(\exists x.(\alpha = (1, x)) \wedge \forall x.(\alpha \neq (1, x)))$	()	5
$\forall x.((x, x) \neq (0, 1))$	$(((\_.0).(\top)))$	1
$\forall x.((x, x) \neq (1, 1))$	()	3
$\forall x.((x, x) \neq (\alpha, 1))$	$(((\_.0).('\_.0 \neq 1)))$	1
$\exists a.\exists b.((\alpha = (a, b)) \wedge \forall x.((x, x) \neq (a, b)))$	$((((\_.0.\_.1)).('\_.0 \neq' \_.1)))$	2

All tests are conducted with a machine with Processor Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 4 Core(s), 8 Logical Processor(s), with Physical Memory (RAM) 8.00 GB, running Racket v8.0[cs] inside WSL2, Ubuntu 18.04.4 LTS.

6.1 Tests from Moiseenko [7]

In this section, we present test cases consistent with those from Moiseenko [7]. Recall that Moiseenko [7] implements universal quantification in terms of negation. We instead implement negation in terms of implication and $\bot$, encoding $\neg G$ as $G \rightarrow \bot$.

First, we run the 15 basic universal quantifier tests from Moiseenko [7], and present the results in Table 1. These tests all use primitive constraints only. We use `(run 1 ...)` for each of the queries and find an answer very quickly (within 22 ms). Moiseenko [7] does not report their OCanren implemention's runtime, and we did not implement their approach in Racket. However, we expect their implementation to be faster than ours.

Results of the form `()` indicate no answer. Results of the form `((var . constraints))` denote an answer with query variable assignments in `var` and additional constraints on these variables in `constraints` (with $\top$ indicating no constraints).

Next, we attempt the graph reachability example and the game-winning example from Moiseenko [7], both presented in Table 2.

The graph test includes a graph with nodes `'(a b c d)` with the relation `edge` denoting directed edges between these nodes. We use implication to describe when a target node is unreachable from a source node.

```
(define-relation (edge x y)
  (disj* (== (cons x y)  (cons 'a 'b))
         (== (cons x y)  (cons 'b 'a))
         (== (cons x y)  (cons 'b 'c))
         (== (cons x y)  (cons 'c 'd))))
(define-relation (reachable x y)
  (disj* (== x y)
         (fresh (z) (edge x z) (reachable z y))))
(define-relation (unreachable x y)
  (implies (reachable x y) (Bottom)))
```

Table 2. Graph Reachability, Winning Example from Moiseenko [7]. We use Greek letters $\alpha, \beta, \ldots$ to represent query variables, and always query for 1 answer.

Query	Result	Time (ms)
`(unreachable c a)`	$(((). (\top)))$	1
`(reachable c a)`	$()$	0
`(unreachable c` α`)`	$(((\_.0).((\_.0 \neq' d) \wedge (\_.0 \neq' c))))$	46
`(unreachable d` α`)`	$(((\_.0).(\_.0 \neq' d)))$	0
`(unreachable` α `a)`	$(((\_.0).(((\_.0 \neq' a) \wedge (\_.0 \neq' b)) \wedge (\_.0 \neq' c))))$	3614
`(unreachable` α `b)`	$(((\_.0).(((\_.0 \neq' a) \wedge (\_.0 \neq' b)) \wedge (\_.0 \neq' c))))$	506
`(unreachable` α `c)`	$(((\_.0).(((\_.0 \neq' a) \wedge (\_.0 \neq' b)) \wedge (\_.0 \neq' c))))$	4061
`(winning c)`	$(((). (\top)))$	7
`(winning d)`	$()$	0
`(winning a)`	We do not expect this query to terminate	>3min
`(winning b)`	We do not expect this query to terminate	>3min

The game-winning test includes the same graph with positions `'(a b c d)` with the relation `edge` denoting directed edges between these positions, indicating movability, where two players take turns. A position is a losing position if it has no moves, or has no winning move. A winning position is one that can move to a losing position (for the next player).

```
(define-relation (winning x)
  (fresh (y) (edge x y)
             (neg (winning y))))
```

The game-winning example is used to demonstrate a non-stratified relation [7]. Our implementation doesn't require stratification, and can run queries over this relation. However, querying `(winning 'a)` or `(winning 'b)` will fail to terminate. For any query where the well-founded semantics[9] for negation would return unknown, our system will fail to terminate.

Finally, we present one more test from Moiseenko [7]. Among the tests from constructive negation, the only test that fails to terminate in a reasonable amount of time (using `(run 1 ..)`) is `(fresh (q) (filter-singleton q (list 1)))`.

```
(define-relation (ifte cond brA brB)
  (disj* (conj cond brA)
         (conj (neg cond) brB)))
(define-relation (filter-singleton xs ys)
  (disj* (conj* (== xs '()) (== ys '()))
         (fresh (x xs2 ys2)
           (== xs (cons x xs2))
           (ifte (fresh (y) (== (list y) x))
                 (== ys (cons x ys2))
                 (== ys ys2))
           (filter-singleton xs2 ys2))))

> (run 1 (q) (filter-singleton q (list 1)))
... ; fails to terminate within 3 minutes
```

6.2 Universal Quantifier and Implication

Table 3 shows additional queries that use universal quantifiers and type constraints. Some of these examples have appeared as examples in previous sections. Notice that the final example is slower than the other examples, due to the presence of a disjunction.

Table 3. Basic Universal Quantifier Tests. We use Greek letters $\alpha, \beta, \ldots$ to represent query variables, and always query for 1 answer.

Query	Result	Time (ms)
$\forall z.\exists x.\exists y.((z, y) = x)$	$(((). (\top)))$	0
$\forall v.((v = 1) \vee ((v \neq 1) \vee (v = 2)))$	$(((). (\top)))$	3
$\exists a.\exists b.\forall v.((v = a) \vee (v \neq b))$	$(((). (\top)))$	4
$\exists a.\exists b.\forall v.(((v = a) \wedge (v \neq b)) \wedge (v \neq v))$	()	0
$((\alpha \in$ '(string?)) $\wedge \forall z.((\alpha \in$ '(symbol? pair? number?)) $\vee ((\alpha = \#t) \vee ((\alpha = \#f) \vee (\alpha =' ())))))$	()	0
$\forall v.((v = 1) \vee ((v \neq 1) \vee (v = 2)))$	$(((). (\top)))$	3
$\forall z.((z = \alpha) \vee (z \neq \beta))$	$(((\_.0 \_.0). (\top)))$	11
$\forall x.\forall y.(((y \neq (a, b)) \vee (x \neq y)) \vee (y = \alpha))$	$((((a.b)). (\top)))$	497

Table 4. Basic Implication Tests. We use Greek letters $\alpha, \beta, \ldots$ to represent query variables, and always query for 1 answer.

Query	Result	Time (ms)
$((\alpha = 1) \rightarrow (\alpha = \beta))$	$(((\_.0 \_.1).('\_.0 \neq 1)))$	0
$\forall a.\forall b.(((b = a) \wedge (a \in$ '(symbol?))) $\rightarrow (b \neq 1))$	$(((). (\top)))$	324
$\forall a.((a = 1) \rightarrow (a \in$ '(symbol?)))	()	1
$\forall x.\forall z.(((x = z) \wedge$ (False z)) $\rightarrow$ (False x))	$(((). (\top)))$	3
$\forall x.\forall y.(((y = (a, b)) \wedge (x = y)) \rightarrow (y = \alpha))$	$((((a.b)). (\top)))$	9212
$(\exists x.($(Loop0) $\vee (x \neq x)) \rightarrow$ (Loop0))	$(((). (\top)))$	2
$(((((((($(Loop1) $\rightarrow$ (Loop2)) $\wedge ($(Loop2) $\rightarrow$ (Loop3))) $\wedge ($(Loop3) $\rightarrow$ (Loop4))) $\wedge ($(Loop7) $\rightarrow$ (Loop4))) $\wedge ($(Loop4) $\rightarrow$ (Loop5))) $\wedge ($(Loop6) $\rightarrow$ (Loop5))) $\rightarrow ($(Loop1) $\rightarrow$ (Loop5)))	$(((). (\top)))$	3209
$(($(Loop x) $\rightarrow$ (Loop8)) $\wedge \exists k.$(Loop k)) $\rightarrow$ (Loop8))	$(((). (\top)))$	448
$\forall x.($(loop x) $\vee$ (neg (loop x)))	We expect that it won't terminate.	>3min
(neg ($\exists x.$(neg ((loop x) $\vee$ (neg (loop x))))))	$(((). (\top)))$	27
((neg (winning a)) $\rightarrow$ (winning b))	$(((). (\top)))$	51
(((winning d) $\rightarrow (\bot)) \rightarrow (\bot)$)	()	3
((winning c) $\rightarrow (\bot)$)	()	19

Table 4 shows the additional queries that use implication. The first few test cases use primitive constraints in their assumptions to test semantic solving. Some of the later tests use user-defined non-terminating relations Loop0 , Loop1 , and others.

The 5th example ($\forall x.\forall y.(((y = (a, b)) \wedge (x = y)) \rightarrow (y = \alpha))$) is particularly interesting. After negating the antecedent, this example becomes equivalent to the final query in Table 3. However, this query takes much longer than the previous, because of the preprocessing (splitting of the negatable and non-negatable components, fruitless branches to search through) we need to do after each relative complement.

Moiseenko [7] expresses universal quantification using negation and existential quantification, but this encoding does not preserve meaning in intuitionistic logic. We provide two examples involving the law of excluded middle (LEM) to demonstrate the difference in behavior: direct LEM ($\forall x.($(loop x) $\vee$ (neg (loop x)))), and the encoded LEM, ($neg(\exists x.(neg($(loop x) $\vee$ (neg (loop x))))))$. The LEM is not an axiom of intuitionistic logic and should not be derivable. This explains why the direct LEM query does not produce an answer

(Table 4 row 9); in fact it does not terminate. However the encoded LEM is derivable in intuitionistic logic, and the encoded LEM query produces an answer (Table 4 row 10). These results are expected given the constructiveness of our system.

The final two examples are extensions of the game-winning examples from Moiseenko [7]. If we assume `a` is a losing position, then we can conclude that `b` is also a winning position. However, even though we know `c` is actually a winning position, our implementation cannot refute `(winning c)` $\rightarrow \bot$ easily. The reason is that the test case solely depends on the analysis of the antecedent, which our implementation analyzes slowly by unfolding, pattern matching and extracting the negatable part of the antecedent. Our implementation analyzes the consequent much faster.

6.3 Evalo Tests

In this section, we use our implementation of universal quantification and implication to reason about a relational interpreter. We use the relational interpreter from Rosenblatt et al. [8].

First, we reason about synthesizing the identity and constant functions. Both functions have consistent behaviour for all inputs. These particular queries are also representable using Eigen[5] and λKanren [4].

```
> (run 1 (x z) (forall (y) () (evalo `(app ,x (quote ,y)) z)))
'(((_.0 (closure '_.0 _.1) _.0) . (\top)))
; cpu time: 57 real time: 57 gc time: 1
> (run 1 (x) (forall (y) () (evalo `(app ,x (quote ,y)) y)))
'(((_.0 (closure (var ()) _.0)) . (\top)))
; cpu time: 285 real time: 285 gc time: 33
```

Next, we test the syntactic solving ability by assuming that a non-terminating expression `omega` is actually an identity function. This query succeeds, but slowly.

```
;; We use de Bruijn indexing to express (λx.x x)(λx.x x)
> (define omega '(app (lambda (app (var ()) (var ())))
                      (lambda (app (var ()) (var ())))))
;; The identity function
> (define idf '(lambda (var ())))
> (run 4 (x) (implies (evalo omega idf)
                      (evalo `(app ,omega ,omega) x)))
'((((closure (var ()) ())) . (\top))
  (((closure (var ()) ())) . (\top))
  (((closure (var ()) ())) . (\top))
  (((closure (var ()) ())) . (\top)))
; cpu time: 15903 real time: 15907 gc time: 1506
```

Note that this query can only be solved via syntactical pattern matching: the program does not terminate, so unfolding `evalo` is fruitless. Thus, the only terminating approach is to use the false assumption that `omega` evaluates to the identity ($\lambda x.x$).

Finally, we present a similar example that uses omega, where the antecedent needs to be unfolded to succeed. However, this query does not succeed in a reasonable time. This example illustrates the asymmetry in how our system treats (consequent) goals and the list of assumptions. Unlike (consequent) goals, which are only unfolded by the semantic solver, assumptions are both unfolded and processed syntactically, but in interleaving search branches that compete for resources.

```
> (run 1 (z) (implies (evalo `(cons ,omega '6) (cons idf 6))
                      (evalo omega z)))
; Takes > 3 mins
```

These examples show why the syntactic matching component is important: it is not sufficient to express implication using negation as in Moiseenko [7] if we wish to reason about potentially non-terminating programs.

[5] http://github.com/webyrd/miniKanren-with-symbolic-constraints

Table 5. Tests on `min` and `sort-boolo` . We use Greek letters $\alpha, \beta, \ldots$ to represent query variables, and always query for 1 answer.

Query	Result	Time (ms)
`(min (list 's) (list))` → `(min-result` α`)`	((((). (⊤))))	2
`(min (list 's 's) (list 's))` → `(min-result` α`)`	(((’s). (⊤)))	28146
∀x.`(sort-boolo (list x #f) (list #f x))`	(((). (⊤)))	1
`( (sort-boolo (list` α `#f) (list #f` α`)` →⊥`) )`	We don’t know if this will terminate.	>3min
∀x.`(sort-boolo (list x #f) (list x #f))`	()	2
(∀x.`(sort-boolo (list x #f) (list x #f))` → (⊥))	(((). (⊤)))	5167
∀x.`(sort-boolo (list #f x #f) (list #f #f x))`	(((). (⊤)))	18
`((sort-boolo (list #f` α `#f) (list #f #f` α`))` → ⊥)	We don’t know if this will terminate.	>3min
∀x.`(sort-boolo (list x #f #f) (list x #f #f))`	()	4
(∀x.`(sort-boolo (list x #f #f) (list x #f #f))` → (⊥))	(((). (⊤)))	35113
`( (sort-boolo (list` α `#f) (list` α `#f))` →⊥`)`	We don’t know if this will terminate.	>3min

6.4 List and Sorting

As a step towards synthesizing provably correct sorting procedures, we test our ability to reason about a sorting relation. With the help of universal quantification and implication, we can prove some of the properties in Table 5.

```
;;; sort the #f into the end, using inserto
(define-relation (inserto x ys xs)
  (conde ((== ys '()) (== xs (list x)))
         ((fresh (y rst rst-inserted)
            (== ys (cons y rst))
            (conde ((=/= y #f) (== xs (cons x ys)))
                   ((== y #f)
                    (== xs (cons y rst-inserted))
                    (inserto x rst rst-inserted)))))))
(define-relation (sort-boolo xs ys)
  (conde ((== xs '()) (== ys '()))
         ((fresh (x rst rst-sorted)
            (== xs (cons x rst))
            (sort-boolo rst rst-sorted)
            (inserto x rst-sorted ys)))))
```

We start with examples that sort lists of two or three elements. Queries using only a universal quantifier run quickly. However, if we use implication to express the same logical meaning, the equivalent queries take significantly longer to run. This performance degradation further demonstrates the asymmetry between processing the assumptions and the (consequent) goal.

Next we try to refute the possibility that `(sort-boolo (list x .. #f) (list x .. #f))` . There is a huge time difference when we change the list length. What’s more, when we are really curious about the counterexample, this negation encoding will still take a very long time. These two groups of examples show the performance deficiency of the current implementation of syntactic solving.

6.5 Using Implication to Compute

To demonstrate the expressiveness of constructive implication, we look at an example that uses implication to carry out a computation inspired by Hallnäs and Schroeder-Heister [3]:

```
;;; find the minimum of two list-encoded Peano numerals
(define-relation (min-result r) (min-result r)) ; intentional non-termination relation
(define-relation (min m n)
  (conj* (implies (== n '()) (min-result '()))
         (implies (== m '()) (min-result '()))
         (forall (m-- n-- r) ()
            (implies (conj*
                       (== m (cons 's m--))
                       (== n (cons 's n--))
                       (implies (min m-- n--) (min-result r)))
                     (min-result (cons 's r))))))
```

Note here `min-result` is encoded as a non-terminating relation. Since `min-result` trivially loops, we can think of it as a placeholder for a result. The actual result will be provided by syntactic solving.

The tests related to min are included in Table 5. Despite the use of non-terminating relations, we obtain results for both $\min(1, 0)$ and $\min(2, 1)$. However, it takes significantly longer to compute $\min(2, 1)$, again due to the asymmetic processing of assumptions described in Section 6.3 and evidenced in Section 6.4.

6.6 Illustrating Constructiveness using the Halting Problem

This next example illustrates another reason to prefer a constructive system, which is to avoid lying to ourselves: we should not be able to solve the halting problem.

Recall in Table 4, we have two examples involving the law of excluded middle (LEM). Here, we replace `loop` with a familiar undecidable relation `halto` , which states that a program will terminate:

```
> (define-relation (halto expr) (fresh (result) (evalo expr result)))
> (run 1 () (forall (expr) ()
              (disj (neg (halto expr)) (halto expr))))
;; The above query will not terminate

> (run 1 () (neg (fresh (expr)
                       (neg (disj (neg (halto expr))
                                  (halto expr))))))
'((() . (\top)))
```

Recall that the direct LEM is not provable in intuitionistic logic, so the direct LEM query will not terminate. The encoded LEM query will eventually terminate, but is incredibly slow. This might be surprising since the original encoded LEM query in terms of `loop` was quite fast. The main reason that replacing `loop` with `halto` leads to such poor performance is that the unfolding of `halto` and `evalo` causes the system to spend more time on fruitless interleaved semantic solving computation before it can find the solution using syntactic solving.

7 LIMITATIONS

There are several limitations in our current implementation of universal quantification and implication.

- **Performance** Since we emphasize expressiveness over performance, many queries involving universal quantification and implication are slow. Moreover, many interesting queries that involve inductive reasoning will fail. We demonstrate in our results the asymmetry of the performance when processing the assumptions and (consequent) goals.
- **Duplicate Solutions** It is possible for the same result to be returned many times, even for simple goals like `(forall (v) (disj (== v a) (=/= v a)))` . Although these repeated results are currently unhelpful, they correspond to alternative ways to construct a proof. We would like the system to eventually produce these proofs.

- **Search is Incomplete** We demonstrate examples where queries do not return a result. Most of these are probably due to performance. However, we conjecture that search is not complete because we don't currently unfold conjuncts fairly in consequent goals.
- **Soundness is not yet proven** We conjecture that the approach is sound, but soundness has not been proven.

8 CONCLUSION AND FUTURE WORK

We present constructive universal quantification and implication in miniKanren. The current implementation is slow, but we are able to successfully run some interesting queries, including stratified examples, to demonstrate the expressive of the approach.

Since the current approach is fairly complex, we are interested in making it possible to check the correctness of the answers. One approach to do so is via *proof generation*, or tracking the steps that the search takes to find an answer. Such proofs can be externally checked and verified. It will also be possible to extract programs from the proof terms as is done in the dependently typed programming setting.

Our goal is to eventually be able to synthesize provably correct programs from universal properties: for example, being able to synthesize a sorting procedure from specification (hence our interest in the examples in Sections 6.3 and 6.5). Such an ability will not only help programmers more easily write code, but also produce an independently checkable proof of correctness.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the helpful comments, and Jonathan Martin, Leon Yao and Ina Jacobson for reading and reviewing the early drafts.

Research reported in this publication was supported by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number OT2TR003435. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

REFERENCES

[1] Krzysztof R Apt and Maarten H Van Emden. 1982. Contributions to the theory of logic programming. *Journal of the ACM (JACM)* 29, 3 (1982), 841–862.

[2] Lars Hallnäs and Peter Schroeder-Heister. 1990. A proof-theoretic approach to logic programming. I. Clauses as rules. *Journal of Logic and Computation* 1, 2 (1990), 261–283.

[3] Lars Hallnäs and Peter Schroeder-Heister. 1991. A proof-theoretic approach to logic programming: II. programs as definitions. *Journal of Logic and Computation* 1, 5 (1991), 635–660.

[4] Weixi Ma, Kuang-Chen Lu, and Daniel P. Friedman. 2020. Higher-order Logic Programming with λKanren. (2020).

[5] Marco Maggesi and Massimo Nocentini. 2020. Kanren Light: A Dynamically Semi-Certified Interactive Logic Programming System. *arXiv preprint arXiv:2007.04691* (2020).

[6] Dale Miller and Gopalan Nadathur. 2012. *Programming with higher-order logic*. Cambridge University Press.

[7] Evgenii Moiseenko. 2019. Constructive negation for minikanren. In *the first miniKanren and Relational Programming Workshop*. 58.

[8] Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. 2019. First-order miniKanren representation: Great for tooling and search. In *the first and Relational Programming Workshop*. 16.

[9] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)* 38, 3 (1991), 619–649.

A APPENDIX

A.1 Proof of the main algorithm of handling Universal Quantifier

We use variable assignment to specify the semantic of a given state. In doing so, we can also specify the semantics of a given miniKanren Goal, which is (computationally) **the** stream of states satisfying the goal. We will model each state as a collection of variable assignments, and thus the semantics of a given goal is just a collection of variable assignments. This is helpful to prove the correctness of our algorithm.

Definition A.1 (Variable Assignment, Semantics of a miniKanren Goal)**.** A variable assignment ϕ is a special substitution that maps variables to literal values in $\mathcal{L}$.

We will inherit the extending notation: $\phi[\mathtt{v} \mapsto a]$ behaves exactly like ϕ except `v` will be substituted by a.

Given a goal `g`, we will denote $[\![\mathtt{g}]\!] = \{\phi : \phi(\mathtt{g}) = \top\}$ to model its semantics, and ϕ is a partial function from the **free variables** of `g` to $\mathcal{L}$.

Relative complements can only be defined via semantic aspect because only in this case is the specification clear. We will first prove the correctness of the algorithm to show that this semantic definition is exactly what we want.

Definition A.2 ((Semantic) Relative Complement)**.** For a given goal `g` with variable `v` inside,

the semantic relative complement of `g` with respect to `v` is exactly the set of variable assignments

$$\{\phi[\mathtt{v} \mapsto n] : \phi[\mathtt{v} \mapsto n] \notin [\![g]\!], \phi \in [\![\texttt{(exists v g)}]\!]\}$$

Looking closely, **this is exactly Definition 3.2**. The complement corresponds to coverage and disjointness. The external consistency is ensured implicitly because the external variables are free variables inside `(exists v g)`.

Proposition A.1 (Correctness of the Algorithm)**.** .

For a family of goals $st_i \in \mathcal{G}$ s.t. $\bigcup_i [\![st_i]\!] = [\![\texttt{(domain} \wedge \texttt{g)}]\!]$,

$$[\![\texttt{forall v domain g}]\!] = [\![\texttt{forall v} \top \texttt{(neg domain)}]\!] \cup \bigcup_i [\![\ \texttt{(exists v } st_i\texttt{)} \wedge \texttt{(forall v (domain} \wedge RC_i\texttt{) g)}\ ]\!]$$

where each RC_i is only semantically defined, and $[\![RC_i]\!]$ is the semantic relative complement of st_i with respect to v,

in other words, $[\![RC_i]\!] = \{\phi[\mathtt{v} \mapsto n] : \phi[\mathtt{v} \mapsto n] \notin [\![st_i]\!], \phi \in [\![\texttt{(exist v } st_i\texttt{)}]\!]\}$

This is proved by expanding definitions on both sides.

Looking closely, this is exactly **how the main algorithm in Section 3 behaves** – the family of st_i is the stream of states (*transformed into goals*) of candidate extraction; the `(forall v ⊤ (neg domain))` corresponds to the relative negation. As long as we have the completeness of solving existential goals, we can ensure the correctness of the algorithm. (Of course inside classical logic, without recursive relation and constructive implication).

Now we need to figure out how to syntactically compute the (semantic) relative complement.

A.2 Correctness of Relative Complements in Algorithm 3.1

Before setting up the correspondence between the algorithmic (syntactical) relative complement and the semantic one, we recall some convention setup.

Convention A.1. We stick to the setup convention ...

- C1.A Recall that we use $\mathcal{L}$ to denote the collection of literal values, inductively defined as a collection of numbers, booleans, strings, symbols, and pairs of literal values.
- C1.B We initially use substitution to indicate variable assignment. Now we use a tuple $\mathcal{L}^n$ to indicate the variable assignment. The order of the tuple is according to the scoping order of a given goal.
- C1.C We use $\mathcal{L}^+ := \bigcup_{n \in \mathbb{N}^{>0}} \mathcal{L}^n$ to denote $\mathcal{L}^n$ for all sorts of n, especially when we don't want to bother with the number of variables.

C1.D We can consider $\mathcal{G}$ to be recursively defined by the following:
given **the (projected) variables** and **constants** from $\mathcal{L}$ as terms t, we have atomic goals/propositions $t_1 = t_2, t_1 \neq t_2, t_1 \in T$ for some (union of) types in T, as well as conjunction, disjunction, and existential quantification.

C1.E Note that, each type T must denote an infinite set; thus we won't have *bool* as a type.
This avoids the potential problems arising from $x \in Bool \wedge x \neq$ `#t` $\wedge\, x \neq$ `#f`.

C1.F $[\![(\cdot)]\!] : \mathcal{G} \to \mathcal{P}(\mathcal{L}^+)$ denotes the collection of values the variable inside $(\cdot)$ can take, and we don't want to specify how many variables are assigned here, so we let codomain be $\mathcal{L}^+$.
We will also use $[\![(\cdot)]\!]$ to denote the mathematical correspondence of the other components we are reasoning about, such as states.

C1.G We use $v_1, .., v_n, x$ to denote the variables appearing inside a goal/proposition P, and $v_1, .., v_n$ are the external logical variables (*mentioned in Definition 3.2*). The x is the quantified variable being considered (*corresponding to the v in Definition 3.2*).
We use $[\![P]\!]_{as/vs}$ to indicate the **evaluation** of goal/propositional P (evaluated to `Top` or `Bottom`) when substituting those variables $v_1, .., v_n, x$ with literal values $a_1, ..a_n, u \in \mathcal{L}$
By this convention, we think of $[\![(\cdot)]\!]$ denoting $n+1$ values, as we stay with the convention taking $n+1$ variables (even though the above definition is the formal one).

C1.H Even though the relative complement is an operation on miniKanren states, we will consider their "propositional correspondents" by translating a state into a proposition (with disjunctions and conjunctions inside).

Convention A.2. .

C2.A (Choice Function): For a given $P \in \mathcal{G}$, since we have $[\![P]\!] = \{(x_1, x_2, .., x_n, x) : P\}$, (where we use conventional set notation to denote the set of values), we can define the choice function f_P^x w.r.t the given (miniKanren) variable x (in P) and the given goal (proposition) P:

$$f_P^x : \mathcal{L}^n \to \mathcal{P}(\mathcal{L})$$
$$f_P^x(a_1, a_2, .., a_n) = \{u : (a_1, a_2, .., a_n, u) \in [\![P]\!]\}$$

Intuitively, this choice function will return a set of all the possible values a miniKanren variable x can take, when other variables inside P are fixed by a_i's.
We omit the superscript x if doing so is unambiguous.

C2.B S^{cwnp} denotes the **c**omplement of S **w**hen S is **n**on-em**p**ty; otherwise $S^{cwnp} = \emptyset$

C2.C We use $(\cdot)^{\not\ni v}$ to indicate a goal/proposition $(\cdot) \in \mathcal{G}$ doesn't have appearances of x inside.

C2.D We use $(\cdot)^{\ni v}$ to indicate a goal/proposition $(\cdot) \in \mathcal{G}$ that has x inside each atomic sub-proposition.

C2.E We use $C_{\mathcal{G}} \subset \mathcal{G}$ to denote all the goals/propositions that use conjunctions as the only connective (*as in Definition 3.4*)
Thus for $C_{\mathcal{G}}$, we can consider each element as inductively constructed by $C_{\mathcal{G}} \ni C = C_1^{\not\ni v} \wedge C_2^{\ni v}$, which are themselves inductively constructed.

C2.F (Field Projection Form) For atomic propositions, we will introduce `car`, `cdr` operators applied to **variables**. Notationally, `(car x)`, `(cdr x)` will be replaced by **field projection form**: `x.car`, `x.cdr`. (*as in Definition 3.5*)
Since this introduces non-canonical forms `(cons x.car x.cdr) = x`, we will exclude `cons` operators from $\mathcal{G}$. All reasoning about pairs will be performed using field projection form.
The advantage is that, for each variable, including **projected variables** (in the form of `v.p.a.t.h`), we can easily decide, by inspecting the projected variable, if it is a term related to the original queried variables; the disadvantage is that it complicates the implementation.

Now we can formalize the semantics of the relative complement and domain filter, using this new notation. Looking closely, we can see this version of the definition actually coincides with the previous one.

Definition A.3 ((Semantic) Relative Complements)**.** We use "$\{..\}_x$" to denote relative complements w.r.t. x. Thus formally speaking, for arbitrary $C \in \mathcal{G}$,

$$\{(a_1, .., a_n, u) : [\![C]\!]_{as/vs}\}_x = \{(a_1, .., a_n, u') : u' \in f_C^x(a_1, .., a_n)^{cwnp}\}$$

Definition A.4 ((Semantic) Domain Filter)**.** We use $\Big|^x$ denotes domain filter away the appearances of x. Thus formally speaking, for arbitrary $C \in \mathcal{G}$,

$$\{(a_1, .., a_n, u) : [\![C]\!]_{as/vs}\}\Big|^x = \{(a_1, .., a_n) : \exists u, [\![C]\!]_{as/vs}\}$$

Domain filter is absent from the main paragraph of the paper because we wrap this concept inside the relative complement. This is a technical detail, useful when, for example, removing the presence of a given variable (*when* `(forall (v) ...)` *returns, we don't want the state to have any information of* `v`). It is an involved concept in the implementation.

Now we postulate some assumptions/properties/axioms. We refer to these as "properties/axioms" because of the considerations of extensibility: to support more atomic constraints, if the following properties are satisfied, then the later proof doesn't need to change as all the proofs are based on the following "axioms".

Proposition A.2 (Some Basic Properties/Axioms to follow when designing (atomic propositions of) $\mathcal{G}$)**.** .

P1.A f_g and $[\![(\cdot)]\!]$ respects (distributes) $\wedge, \vee$
i.e. $f_{g_1 \wedge g_2}(\cdot) = f_{g_1}(\cdot) \cap f_{g_2}(\cdot)$, ...

P1.B we have a function $\texttt{DomainEnforce}^x(\cdot) : \mathcal{G}^{\ni v} \to \mathcal{G}^{\ni v}$ s.t.

(a) $\texttt{DomainEnforce}^x(A \wedge B) \iff \texttt{DomainEnforce}^x(A) \wedge \texttt{DomainEnforce}^x(B)$

(b) if $f_{\texttt{DomainEnforce}^x(g^{\ni v})}(a_1, .., a_n) \neq \emptyset$,
then $f_{\texttt{DomainEnforce}^x(g^{\ni v})}(a_1, .., a_n)^c = f_{\overline{\texttt{DomainEnforce}^x(g^{\ni v})}}(a_1, .., a_n)$ for atomic proposition $g^{\ni v}$

P1.C (Non-Empty Precondition)
We have a function $\texttt{DomainEnforce} : \mathcal{G}^{\ni v} \to \mathcal{G}^{\not\ni v}$ s.t.

(a) $\forall\{a_i\}, (a_1, .., a_n) \in [\![\texttt{DomainEnforce}(C^{\ni v})]\!] \implies f_{C^{\ni v}}(a_1, .., a_n) \neq \emptyset$
(Note, $C^{\ni v}$ has to be satisfiable)

(b) For $C = C^{\not\ni v} \wedge C^{\ni v}$, $[\![\texttt{DomainEnforce}(C^{\ni v})]\!] \supseteq [\![C^{\ni v}]\!] \supseteq [\![C]\!]$

P1.D $f_{g^{\not\ni v}}(a_1, .., a_n) = \mathcal{L}$ or $f_{g^{\not\ni v}}(a_1, .., a_n) = \emptyset$

P1.E $\{(a_1, .., a_n, u) : [\![C]\!]_{as/vs}\}\Big|^x = \{(a_1, .., a_n) : f_C(a_1, .., a_n) \neq \emptyset\}$

Here $\texttt{DomainEnforce}, \texttt{DomainEnforce}^x$ are just **the extensional/axiomatic view of making pairo explicit mentioned in** Algorithm 3.1. In other words, computationally, these two functions will just make sure `pairo` constraints are explicit.

Remark A.1 (Intuition about $\texttt{DomainEnforce}, \texttt{DomainEnforce}^x$)**.** .

The definition of $\texttt{DomainEnforce}^x$ inside Item P1.Bb comes from the naive expectation $f_{\overline{g}}(\cdot) = f_g(\cdot)^c$ – i.e. we can lift the negation of formula onto the set-level. However, it requires the second thought on the negation operation $\overline{(\cdot)}$: for example, intuitively speaking we will design the negation as $\overline{\texttt{x.car = 1}} \equiv \texttt{x.car} \neq 1$. However, this doesn't coincide with the intuition of the semantic complement: if `car x = 1` doesn't hold, it could also be the case that `x` is not even a pair in the first place.

The reason this happens is that, projection (car and cdr), when applied to a variable, actually has some precondition on the variable to begin with. That's the job of $\texttt{DomainEnforce}^x$ – making sure this precondition is explicit. We will ensure when taking (syntactic) negation, we will start with $\texttt{DomainEnforce}^x(\texttt{x.car = 1}) \equiv$ (`x.car = 1` $\wedge$ `x` $\in$ `Pair`). The naive syntactic complement will also consider the case where `x` is not even a pair.

The same holds true for `DomainEnforce`, as we don't want the choice function to fix the value of other variables nonsensically. For example, when we have a goal/proposition like `x = a.car` it is not reasonable to let a be a number. Thus applying `DomainEnforce` to this goal/proposition will lead to an explicit "`a` $\in$ `Pair`". What's more, `DomainEnforce` never imposes "new" information; that is the reason we can have Item P1.Cb.

Thus $\texttt{DomainEnforce}, \texttt{DomainEnforce}^x$ are just functions that are responsible for excluding the "invalid" sentence (a sentence with no proper domain specified and thus ambiguous). Their difference is that $\texttt{DomainEnforce}^x$ handles the type constraint of x, while `DomainEnforce` handles the type constraints of the remaining variables.

Proposition A.3 (Lemma only relies on the postulated properties/axioms)**.** Most of the following are just proved by induction on the structure of $C_{\mathcal{G}}$ and their components. (Note that, each following $C, C_1, C_2, ..$, no matter $(\cdot)^{\not\ni v}$ or $(\cdot)^{\ni v}$ all belong to $C_{\mathcal{G}}$).

L3.A $f_{C^{\not\ni v}}(a_1, .., a_n) = \mathcal{L}$ or $\emptyset$
L3.B if $f_{\texttt{DomainEnforce}^x(C^{\ni v})}(a_1, .., a_n) \neq \emptyset$,
then $f_{\texttt{DomainEnforce}^x(C^{\ni v})}(a_1, .., a_n)^c = f_{\overline{\texttt{DomainEnforce}^x(C^{\ni v})}}(a_1, .., a_n)$
L3.C if the logical statement "$C_1^{\not\ni v} \rightarrow \texttt{DomainEnforce}(C_2^{\ni v})$" is valid, then
L3.C.A $f_{C_1^{\not\ni v}}(a_1, .., a_n) \neq \emptyset \implies f_{C_2^{\ni v}}(a_1, .., a_n) \neq \emptyset$
L3.C.B $f_{C_1^{\not\ni v}}(a_1, .., a_n) \neq \emptyset \iff f_{C_1^{\not\ni v} \wedge C_2^{\ni v}}(a_1, .., a_n) \neq \emptyset$

Proof. .
Item L3.A is proved by induction on $C^{\not\ni v}$, with the help of Item P1.A and Item P1.D.
Item L3.B is proved by induction on $C^{\ni v}$ with the help of Item P1.Ba and Item P1.Bb.
To prove Item L3.C.A, we know $f_{C_1^{\not\ni v}}(a_1, .., a_n) \neq \emptyset$ means there exists u,such that $[\![C_1]\!]_{as/vs} = TOP$. Since the logical statement "$C_1^{\not\ni v} \rightarrow \texttt{DomainEnforce}(C_2^{\ni v})$" is valid, we know $[\![\texttt{DomainEnforce}(C_2^{\ni v})]\!]_{as/vs} = TOP$. What's more, u and x are absent from $\texttt{DomainEnforce}(C_2^{\ni v})$. Thus that means $(a_1, a_2, .., a_n) \in [\![\texttt{DomainEnforce}(C_2^{\ni v})]\!]$, by Item P1.C, we know $f_{C_2^{\ni v}}(a_1, .., a_n) \neq \emptyset$.
Item L3.C.B is proved by Item L3.C.A and Item L3.A and Item P1.A.

□

Proposition A.4 ((Syntactic) Relative Complement)**.** For arbitrary $C_1^{\not\ni v}$ and $C_2^{\ni v} \in \mathcal{C}_{\mathcal{G}}$ where we assume

(1) $C_1^{\not\ni v} \wedge C_2^{\ni v}$ satisfiable,
(2) "$C_1^{\not\ni v} \rightarrow \texttt{DomainEnforce}(C_2^{\ni v})$" is valid
(3) "$C_2^{\ni v} \iff \texttt{DomainEnforce}^x(C_2^{\ni v})$" is valid

then we have

$$\{(a_1, .., a_n, u) : [\![C_1^{\not\ni v} \wedge C_2^{\ni v}]\!]_{as/vs}\}_x = \{(a_1, .., a_n, u) : [\![C_1^{\not\ni v} \wedge \overline{C_2^{\ni v}}]\!]_{as/vs}\}$$

Proof. By definition, we only need to verify the second equality of the following

$$\begin{aligned}\{(a_1, .., a_n, u) : [\![C_1^{\not\ni v} \wedge C_2^{\ni v}]\!]_{as/vs}\}_x &= \{(a_1, .., a_n, u) : u \in (f_{C_1^{\not\ni v} \wedge C_2^{\ni v}}(a_1, .., a_n))^{cwnp}\} \\ &=^? \{(a_1, .., a_n, x) : [\![C_1^{\not\ni v} \wedge \overline{C_2^{\ni v}}]\!]_{as/vs}\}\end{aligned}$$

Thus it is sufficient to prove in this context,
for arbitrary $a_1, .., a_n, u \in \mathcal{L}$,

$$u \in (f_{C_1^{\not\ni v} \wedge C_2^{\ni v}}(a_1, .., a_n))^{cwnp} \iff [\![C_1^{\not\ni v} \wedge \overline{C_2^{\ni v}}]\!]_{as/vs}$$

When $(f_{C_1^{\not\ni v} \wedge C_2^{\ni v}}(a_1, .., a_n)) = \emptyset$,
and due to Item L3.A:
If $f_{C_1^{\not\ni v}}(a_1, .., a_n) = \mathcal{L}$, then $f_{C_2^{\ni v}}(a_1, .., a_n) = \emptyset$, but it contradicts Item L3.C.A.
If $f_{C_1^{\not\ni v}}(a_1, .., a_n) = \emptyset$, then
$u \in \emptyset \iff False \iff {}^{\dagger}[\![C_1^{\not\ni v}]\!]_{as/vs} \iff [\![C_1^{\not\ni v} \wedge \overline{C_2^{\ni v}}]\!]_{as/vs}$
The † can be proved by expanding the definition.
When $(f_{C_1^{\not\ni v} \wedge C_2^{\ni v}}(a_1, .., a_n)) \neq \emptyset$,
Then $(f_{C_1^{\not\ni v}}(a_1, .., a_n)) = \mathcal{L}$, and then

$$\begin{aligned}u \in (f_{C_1^{\not\ni v} \wedge C_2^{\ni v}}(a_1, .., a_n))^{cwnp} &\iff u \in (f_{C_2^{\ni v}}(a_1, .., a_n))^c \\ &\iff u \in (f_{\overline{C_2^{\ni v}}}(a_1, .., a_n)) && \text{by } \textit{Item L3.B} \\ &\iff [\![\overline{C_2^{\ni v}}]\!]_{as/vs} && \text{by definition of } f \\ &\iff [\![C_1^{\not\ni v} \wedge \overline{C_2^{\ni v}}]\!]_{as/vs} && \text{since } [\![C_1^{\not\ni v}]\!]_{as/vs} = TOP\end{aligned}$$

□

This part directly refers to syntactic relative complements in Proposition 3.3 and all the preprocessing we have done to satisfy the assumption of the proposition.

Proposition A.5 ((Syntactic) Domain Filter)**.** .

If the logical statement "$C'^{\not\ni v} \rightarrow \texttt{DomainEnforce}(C^{\ni v})$" is valid, then

$$
\begin{aligned}
&\{(a_1, .., a_n, u) : [\![C^{\ni v} \wedge C'^{\not\ni v}]\!]_{as/vs}\}\Big|^{x} \\
&= \{(a_1, .., a_n, u) : f_{C^{\ni v} \wedge C'^{\not\ni v}}(a_1, .., a_n) \neq \emptyset\} && \text{By Item P1.E} \\
&= \{(a_1, .., a_n) : f_{C'^{\not\ni v}}(a_1, .., a_n) \neq \emptyset\} && \text{By Item L3.C.B} \\
&= \{(a_1, .., a_n) : \exists u, (a_1, .., a_n, u) \in [\![C'^{\not\ni v}]\!]\} && \text{By definition of } f^{x}_{C'^{\not\ni v}} \\
&= \{(a_1, .., a_n) : \exists u, [\![C'^{\not\ni v}]\!]_{as/vs}\} && \text{By definition of } [\![(\cdot)]\!]_{as/vs} \\
&= \{(a_1, .., a_n) : [\![C'^{\not\ni v}]\!]_{as/vs}\} && x \text{ and } u \text{ are not appearing}
\end{aligned}
$$

This syntactic domain filter justified why, in the relative complement algorithm, we can simply remove the unmentioned variable directly. The preprocessing directly refers to the precondition of this proposition.

This proposition also justifies why the inner existential quantifiers of a universal quantifier can be directly ignored after proper processing (including explicit pairo and field projection form). The main reason can be seen in the item P1.C.

Proposition A.6 (Unsatisfiability Checking)**.** .

If the logical statement "$C'^{\not\ni v} \rightarrow \texttt{DomainEnforce}(C^{\ni v})$" is valid,
take $a_1, .., a_n$ as arbitrary,
where we denote $C = C^{\ni v} \wedge C'^{\not\ni v}$, and there are $v_1, .., v_n, v$ as logical variables,
and $[\![C^{\ni v} \wedge C'^{\not\ni v}]\!]_{\vec{a}/\vec{v}}$ as the resulting goal of substituting $v_1, .., v_n$ by $a_1, .., a_n$.
then

$$
\begin{aligned}
[\![C^{\ni v} \wedge C'^{\not\ni v}]\!]_{\vec{a}/\vec{v}} \text{ is unsatisfiable} &\iff f_{C^{\ni v} \wedge C'^{\not\ni v}}(a_1, .., a_n) = \emptyset && \text{By Item P1.E} \\
&\iff f_{C'^{\not\ni v}}(a_1, .., a_n) = \emptyset && \text{By Item L3.C.B} \\
&\iff [\![C'^{\not\ni v}]\!]_{\vec{a}/\vec{v}} = BOTTOM \\
&\iff [\![\overline{C'^{\not\ni v}}]\!]_{\vec{a}/\vec{v}} = TOP
\end{aligned}
$$

This part directly refers to how relative negation is computed in Proposition 3.6.

Take the relative complement algorithm as the example. The Algorithm 3.1 directly corresponds to Proposition A.4.

All we need to prepare is to (1) make sure we are dealing with conjunction-only (by splitting the stream) just like we are dealing with $C = C^{\ni v} \wedge C^{\not\ni v}$ (*as Definition 3.4*) ,(2) sort out the atomic goals/propositions inside intermediate state/goals, find out which belongs ${\mathcal{C}_{\mathcal{G}}}^{\not\ni v}$ and ${\mathcal{C}_{\mathcal{G}}}^{\ni v}$ (*as Proposition 3.3*), and (3) make $C^{\not\ni v} \rightarrow \texttt{DomainEnforce}(C^{\ni v})$ valid by dealing with $C \wedge \texttt{DomainEnforce}(C^{\ni v})$ instead. More concretely in our framework, we are dealing with explicitly mentioning `pairo` . *(This is sound and complete due to Item P1.Cb and thus* $C \iff C \wedge \texttt{DomainEnforce}(C^{\ni v})$*).*