Constructive Negation for MiniKanren

EVGENII MOISEENKO, Saint Petersburg State University and JetBrains Research, Russia

We present an extension of MiniKanren with the negation operator based on the method of *constructive negation*. The idea of this method is to constructively build a stream of answers for the negated goal by collecting and negating individual answers to the positive version of the goal. As we demonstrate on the series of examples constructive negation suits to pure logical nature of MiniKanren: the relations involving the negation operator still can be "run" in various directions.

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

Additional Key Words and Phrases: relational programming, constructive negation, OCanren

1 INTRODUCTION

MiniKanren [Byrd 2010; Friedman et al. 2005] is a minimalistic domain-specific language which brings features of logic programming into a host language. Typical MiniKanren implementation introduces only a few operators: conjunction, disjunction, unification (which can be seen as equality constraint) and fresh variable introduction (existential quantification). Although this basis constitutes a Turing-complete language, in practice there are some cases when the availability of negative reasoning is desirable.

For example, consider the following problem. Suppose we want to write a program which removes the first occurrence of a given element from a list. In order to do that in MiniKanren we have to first define a ternary relation `remove` which binds the desired element, original list, and the same list after the deletion. Substituting the first argument of the relation with some element `e`, the second argument with some list `xs` and the third argument with a free variable `q` gives us a *goal* which is a proof search procedure. When passed to the `run` function, the goal will produce a lazy stream of *answers*. Each answer is represented by substitution which binds free variables to some terms. In the case of `remove`, we would expect a single answer, which binds `q` to the list, equal to `xs`, except that the first occurrence of `e` is removed.

The code on Listing 1 demonstrates a possible implementation of `remove`. It consists of three disjuncts, which represent three different cases. First, if the original list is empty, then the resulting list should also be empty. If the original list is not empty and its head is equal to the given element, then the resulting list should be equal to the tail. Finally, in the case when the head is not equal to the given element, the resulting list should be equal to the original, from the tail of which the occurrence of the element `e` is deleted.

Given the definition of `remove` from Listing 1, the following query `run (remove 2 [1;2;3] q)` will wrongly return two answers: `q =[1;3]` and `q =[1;2;3]` . The redundant (and, indeed, incorrect) answer `q =[1;2;3]` arose because of the third disjunct from `remove` definition, which always succeeds and thus generates a copy of the original list. We can prevent this behavior using *disequality constraint*, yet another primitive, which some MiniKanren implementations provide. Adding constraint `x ≢ e` to the third case makes all disjuncts mutually

Author's address: Evgenii Moiseenko, e.moiseenko@2012.spbu.ru, Saint Petersburg State University, Saint Petersburg , JetBrains Research, Russia.

disjoint. With the fixed definition of remove (Listing 2), the query given above returns the single answer q =[1;3] as expected.

```
let remove e xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs′) (
    x ≡ e ∧
    xs ≡ x::xs′ ∧
    ys ≡ xs′
  ) ∨
  fresh (x xs′ ys′) (
    xs ≡ x::xs′ ∧
    ys ≡ x::ys′ ∧
    remove e xs′ ys′
  )
```

Listing 1. A flawed definition of remove relation

```
let remove e xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs′) (
    x ≡ e ∧
    xs ≡ x::xs′ ∧
    ys ≡ xs′
  ) ∨
  fresh (x xs′ ys′) (
    x ≢ e ∧
    xs ≡ x::xs′ ∧
    ys ≡ x::ys′ ∧
    remove e xs′ ys′
  )
```

Listing 2. The correct definition of the remove relation

```
let remove p xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs′) (
    p x ∧
    xs ≡ x::xs′ ∧
    ys ≡ xs′
  ) ∨
  fresh (x xs′ ys′) (
    ¬(p x) ∧
    xs ≡ x::xs′ ∧
    ys ≡ x::ys′ ∧
    remove p xs′ ys′
  )
```

Listing 3. A generalized version of the remove relation

A disequality constraint represents a very limited form of negation, which is often not sufficient. Imagine that we want to generalize remove, so that instead of taking an element, it takes a predicate p and removes the first element of the list which satisfies the predicate (Listing 3). In order to do that and avoid similar pitfalls as in our first attempt to define remove, we need to ensure that the third disjunct succeeds only when the predicate p fails on the head element of xs. Thus we need a general *negation operator*. Unfortunately, none of the existing MiniKanren implementations provide this feature.

In the world of Prolog, negative reasoning is usually implemented using so-called *negation as failure* approach. Under this rule, a goal ¬p succeeds whenever p fails, and it fails whenever p succeeds. Unfortunately, negation as failure is unsound if the negated goal p contains free variables. For example, the query run (q ≡ 0) ∧ ¬(q ≡ 1) succeeds, although run ¬(q ≡ 1) ∧ (q ≡ 0) fails. The failure in the last case is due to the fact that at the time of negation execution the variable q is free, thus q ≡ 1 succeeds, and conversely ¬(q ≡ 1) fails. This example demonstrates that negation as failure goes against the philosophy of MiniKanren, which positions itself as a pure relational language without non-logical features.

In this work we present an implementation of the negation operator based on the method of *constructive negation*. It overcomes several shortcomings of negation as failure and provides a more expressive form of negative reasoning than disequality constraints. The idea of this method is to constructively build a stream of answers for the negated goal. It is achieved by first collecting all answers to the positive version of the goal and then negating them. The constructive negation approach while being an improvement over negation as failure still has its own drawbacks. For example, applying the negation operator to the goal that has infinitely many answers results in a non-terminating computation.

Although constructive negation was first proposed as an extension of Prolog, to the best of our knowledge our work is the first attempt to adapt it for MiniKanren. We have developed a prototype implementation for OCanren [Kosarev and Boulytchev 2018], a MiniKanren dialect embedded in OCaml. Like the rest of

MiniKanren, our version of constructive negation is developed in a pure functional style using persistent data structures. This makes it different from the earlier implementations for Prolog.

Our paper is structured as follows. In Section 2 we give more examples of constructive negation usage. Section 3 describes our implementation. Next, Section 4 presents the evaluation of the negation on a series of examples. In Section 5 we discuss the limitations of our approach and directions for future work. Section 6 reviews related works. The final Section 7 concludes.

2 MOTIVATING EXAMPLES

In this section we give further motivation for adding the negation into relational programming. We present several examples of how the negation can be used.

2.1 Relational If-then-else

In the Prolog one can simulate the conditional if-then-else operator using so-called *soft cut* [Naish 1995]. The behavior of the soft cut `c` $\rightarrow^*$ `t` ; `e` can be described as follows:

- if the goal `c` succeeds (i.e. produces at least one answer) then the result of `c` $\rightarrow^*$ `t` ; `e` is equivalent to `c ∧ t`;
- if the goal `c` fails (i.e. produces no answers at all) then the result of `c` $\rightarrow^*$ `t` ; `e` is equivalent to `e`.

The soft cut is an example of a non-relational feature. Such features usually do not compose well in the sense that they are sensitive to the order in which they appear in a program. For example, consider the goal (`c` $\rightarrow^*$ `t` ; `e`) ∧ `g`, and assume that `c ∧ g` always fails regardless of the order of conjuncts. Then if `c` succeeds the result of the above goal will be equivalent to `c ∧ t ∧ g` and thus it will fail. Suppose we reorder the conjuncts as follows: `g ∧ (c` $\rightarrow^*$ `t` ; `e`). Now the goal `c`, when executed after `g`, certainly fails, and thus the result of the whole goal will be equivalent to `g ∧ e` which does not fails necessarily. One can see that the simple reordering of subgoals in the program can lead to the different results.

With the help of constructive negation if-then-else can be simply expressed as follows:

```
let ifte c t e =
  (c ∧ t) ∨ (¬ c ∧ e)
```

The behavior of if-then-else defined in this way subsumes the behavior of the soft cut. That is, every answer to the query `run (c` $\rightarrow^*$ `t` ; `e`) is also an answer to the query `run (ifte c t e)`. Moreover, `ifte` is not sensitive to the order of subgoals in the program.

2.2 Classical Implication and Universal Quantification

With the negation added to the language, one can easily express other connectives of the classical first-order logic, namely the implication and universal quantification[1], using the well-known equivalences.

```
let (⇒) : goal → goal → goal =
  fun g₁ g₂ → ¬ g₁ ∨ (g₁ ∧ g₂)

let forall : ('a → goal) → goal =
  fun g → ¬ fresh (x) (¬ g x)
```

However, we should make a few remarks here. It is well known that the search implemented in conventional MiniKanren is complete, meaning that every answer to an arbitrary query will be found eventually. In the presence of constructive negation (and thus implication and universal quantification defined through negation)

[1] It is easy to see that `c ⇒ t` is equivalent to `ifte c t succ`

the search becomes incomplete as we will later see. Moreover, constructive negation is computationally heavy and thus the double usage of it, as in the definition of `forall`, can be inefficient in some cases.

Despite all this trouble we have found that the above definitions are still useful. Some of the previous MiniKanren implementations introduced *eigen* variables, adopted from λProlog [Miller and Nadathur 2012]. Eigen variables act as a universally quantified variables. Yet, to the best of our knowledge, there is no sound implementation of eigen variables with the support of disequality constraints. We observed that our implementation of universal quantifications through double negation works nicely with disequalities (we give some examples in the Section 4).

2.3 Graph Unreachability Problem

One of the classical examples of negation application in logic programming is a problem of checking whether one node of the graph is unreachable from the another [Przymusinski 1989]. The code on Listing 4 defines binary relation `edge`, which binds pairs of nodes in graph, connected by some edge, and binary relation `reachable`, which is nothing more than a transitive closure of the `edge` relation. Then the relation `unreachable` is simply negation of `reachable`.

```
let edge x y =
  (x, y) ≡ ('a', 'b') ∨
  (x, y) ≡ ('b', 'a') ∨
  (x, y) ≡ ('b', 'c') ∨
  (x, y) ≡ ('c', 'd')

let reachable x y =
  x ≡ y ∨
  fresh (z) (
    edge x z ∧ reachable z y
  )

let unreachable x y =
  ¬(reachable x y)
```

Listing 4. Unreachability in a graph

Given this definition the query `run unreachable 'c' 'a'` will succeed. A knowledgeable reader might notice that constructive negation is not necessary in this case because negation as failure will deliver the same result. But the query `run unreachable 'c' q` will fail under negation as failure because of the free variable `q` which will appear under the negation. However constructive negation will succeed delivering the constraint `q ≢ 'd'`.

2.4 Unreachability in Labeled Transition Systems

One can consider a special kind of graphs — *labeled transition systems* [Baier and Katoen 2008]. Labeled transition system is defined by a set of states S, a set of labels L and a ternary transition relation $R \subseteq S \times L \times S$. By existential quantification over labels one can then obtain a binary relation. Taking its transitive closure gives the reachability relation. The negation of the reachability relation can be used to check that some state s' is not reachable from the initial state s. The Listing 5 shows an encoding of an abstract labeled transition system in OCanren.

Labeled transition systems are often used to describe the behavior of imperative languages. Although the naive encoding of transition relation in OCanren with simple enumeration of reachable states is often not tractable for checking reachability (or unreachability) in huge state spaces arising in practical imperative programs, it still can be used, for example, for the task of prototyping the semantics of such languages.

```
module type LTS = sig
  type state
  type label

  val transition : state → label → state → goal
end

module LTSExt (T : LTS) = struct

  let reachable : T.state → T.state → goal =
    fun s s'' →
      s ≡ s''
      ∨
      fresh (l s') (
        T.transition s l s' ∧
        reachable s' s''
      )

  let unreachable : T.state → T.state → goal =
    fun s s' →
      ¬(reachable s s')
end
```

Listing 5. Unreachability in a labeled transition system

3 IMPLEMENTATION

In this section we present our implementation of constructive negation. We start with the general ideas behind the method (Section 3.1). We describe how the constructive negation behaves in concrete examples, starting from trivial ones and moving to more sophisticated. During this presentation, we will observe, that in order to implement constructive negation, we need a solver for universally quantified disequality constraints. We will show that such solver can be implemented on top of existing MiniKanren disequality solver with just a few modifications (Section 3.2). In the Section 3.3, we describe how the OCanren search can be extended to support constructive negation. We will also discuss how negation interacts with recursion and present the notion of *stratification* (Section 3.4).

3.1 General Ideas

Constructive negation is based on the following idea: given a goal ¬g, one can construct an answer for this goal by collecting all answers to its positive version g and then taking their complementation. In order to do that, a notion of "negation" of an answer is needed. Since each answer can be matched to some logical formula [Przymusinski 1989; Stuckey 1991], a "negation" of an answer corresponds to the logical negation of this formula.

Example 1. Consider the goal `¬(q ≡ 1 ∧ r ≡ 2)`. Its positive version `(q ≡ 1) ∧ (r ≡ 2)` has single answer, a substitution {`q` ↦ `1`, `r` ↦ `2`} which corresponds to the formula $q = 1 \wedge r = 2$. By negating this formula we obtain $q \neq 1 \vee r \neq 2$. This formula still can be represent by a single substitution {`q` ↦ `1`, `r` ↦ `2`}. However, we now treat this substitution differently, as a *disequality constraint*.

Some MiniKanren implementations (including OCanren) already have the support for disequality constraints. A programmer can use them with the help of ≢ primitive, as we have seen in the `remove` example (Listing 2). Usually, the support for disequalities is implemented as follows.

- A current state is maintained during the search. The state consists of a substitution, which represents positive information, and a disequality constraint store, which represents negative information. A constraint store can be implemented simply as a list of substitutions, or as a more efficient data structure.
- Each time a subgoal of the form `t ≢ u` is encountered during the search, its satisfiability in current substitution is checked. If it is satisfiable, then disequality is added to the constraint store.
- Each time a subgoal of the form `t ≡ u` is encountered during the search, the current substitution is refined by the result of unification of terms `t` and `u`. Then the satisfiability of disequality constraints is rechecked in the refined substitution.

Various optimizations can be applied to the scheme above. For example, there is no need to recheck every disequality in the store after each unification. We will not discuss these optimizations here, as they are irrelevant to our goals.

Unfortunately, as the next example illustrates, disequalities presented above are not sufficient to implement constructive negation.

Example 2. Consider a goal ¬(**fresh** `(x) (q ≡ (x, x))`), which states that `q` should not be equal to some pair of identical terms. The subgoal **fresh** `(x) (q ≡ (x, x))` succeeds, delivering the substitution {`q` ↦ `(x, x)`}. Because the variable `x` occurs under **fresh**, the corresponding formula is existentially quantified: $\exists x.\, q = (x, x)$. By the negation of this formula we obtain $\forall x.\, q \neq (x, x)$. This formula differs from disequality formula from example 1 as it contains universally quantified variable `x`.

Thereby, in order to support the negation of goals, containing **fresh**, we need to extend disequality constraint solver, so it can check the satisfiability of *universally quantified disequality constraints* in the form $\forall \overline{x}.\, t \neq u$ [2] [Chan 1988; Stuckey 1991]. Later, in Section 3.2 we will show how it can be done, for now let us assume we have such a solver.

We took care about **fresh** under negation. It led us to a more complicated representation of the state. During the search we maintain a pair of a current substitution and a universally quantified disequality constraint store. But now an interesting question arises: is this representation closed under negation? If we perform negation one more time, will we obtain a finite number of states in a similar form?

Lucky for us, it is the case. In order to verify it, let us consider a logical formula which corresponds to the representation of the state:

$$\exists \overline{x}.\, \left(\bigwedge_i (v_i = t_i) \wedge \bigwedge_j \forall \overline{y_j}.\, \bigvee_k (w_{jk} \neq u_{jk}) \right) \tag{1}$$

Here v_i and w_{jk} denote some variables, t_i and u_{jk} denote some terms. Existentially quantified variables $\overline{x}$ correspond to the variables occurred under **fresh**. The left conjunct corresponds to the substitution, the right conjunct corresponds to the constraint store. The constraint store itself is represented as a conjunction of individual universally quantified disequalities. As we have seen in example 1, each disequality corresponds to a

[2] $\overline{x}$ notation denotes a vector of variables, t and u are terms that may or may not contain variables from $\overline{x}$

disjunction of individual inequalities over variables[3]. Besides existential variables $\overline{x}$ and universal variables $\overline{y_j}$, there are free variables $\overline{q}$ that may occur in the formula (such as variables q and r in the examples 1, 2).

By logical negation of the above formula, we get the following:

$$\forall \overline{x}.\left(\bigvee_i (v_i \neq t_i) \vee \bigvee_j \exists \overline{y_j}.\ \bigwedge_k (w_{jk} = u_{jk})\right) \tag{2}$$

The left disjunct, which corresponded to the substitution in the original formula, now became a disequality constraint. Looking at the right disjunct, we can see that each disequality has transformed into a substitution. However, there is one subtlety here. Variables w_{jk} may be universally quantified, that is $w_{jk} \in \overline{x}$ for some j, k. Moreover, the terms u_{jk} may contain universally quantified variables as well, $Vars(u_{jk}) \subseteq \overline{x}$ for some j, k. In the section 3.2 we will show, that each disjunct $\exists \overline{y_j}.\ \bigwedge_k (w_{jk} = u_{jk})$ is either unsatisfiable or could be rewritten as $\exists \overline{y_j}.\ \bigwedge_{k^*} (w^*_{jk^*} = u^*_{jk^*})$, such that universally quantified variables do not occur among variables $w^*_{jk^*}$ or in terms $u^*_{jk^*}$ [Liu et al. 1999].

Taking this into account, we can rewrite the formula 2 as follow:

$$(\bigvee_j \exists \overline{y_j}.\ \bigwedge_{k^*} (w^*_{jk^*} = u^*_{jk^*})) \vee \forall \overline{x}.\ \bigvee_i (v_i \neq t_i) \tag{3}$$

In the obtained formula each disjunct corresponds to one state in the form, similar to the given in formula 1. In the left disjunct, each sub-disjunct corresponds to a substitution with an empty disequality constraint, the right disjunct corresponds to the single universally quantified disequality constraint with an empty substitution. Thereby, the proposed representation of states is closed under negation.

One can perform further manipulations on the formula 3. Given that equivalence $a \vee \neg b = a \wedge b \vee \neg b$ holds in classical logic, we can rewrite formula 3 in the following way:

$$(\exists \overline{x}.\ \bigwedge_i (v_i = t_i) \wedge \bigvee_j \exists \overline{y_j}.\ \bigwedge_{k^*} (w^*_{jk^*} = u^*_{jk^*})) \vee \forall \overline{x}.\ \bigvee_i (v_i \neq t_i) \tag{4}$$

The latter transformation, while vacuous from the logical point of view, could improve the performance of the search in practice. It follows from the fact, that the subpart $\exists \overline{x}.\ \bigwedge_i (v_i = t_i)$ of the formula extends each substitution with additional mappings, thus delivering more positive information. If the negation constitutes a subpart of some larger goal, this positive information could lead to the earlier failure during the search.

Finally let us consider the negation in general case. A goal can be matched to a logical formula in the following way. Each answer to the goal corresponds to the state which itself corresponds to a logical formula in the form similar to one in formula 1. Goal can have multiple answers (even an infinite number). In the corresponding logical formula these answers will be connected by the disjunction:

$$\bigvee_n \left(\exists \overline{x_n}.\left(\bigwedge_i (v_{ni} = t_{ni}) \wedge \bigwedge_j \forall \overline{y_{nj}}.\ \bigvee_k (w_{njk} \neq u_{njk})\right)\right) \tag{5}$$

The negation of this formula after an application of the transformations described above will become:

$$\bigwedge_n \left((\exists \overline{x_n}.\ \bigwedge_i (v_{ni} = t_{ni}) \wedge \bigvee_j \exists \overline{y_{nj}}.\ \bigwedge_{k^*} (w^*_{njk^*} = u^*_{njk^*})) \vee \forall \overline{x_n}.\ \bigvee_i (v_{ni} \neq t_{ni})\right) \tag{6}$$

[3] We will give further explanation in section 3.2

In this way, we have obtained the result of the negation of the goal as a conjunction of the negation of individual answers to the positive version of the goal. However, one of the pitfalls of this construction is that the process will not terminate if the positive version of the goal has an infinite number of answers. Thus, in the general case, it makes the OCanren search incomplete for the goals involving negation.

3.2 Constraint Solver, Formally

In this section we will formally define the satisfiability of universally quantified disequality constraints and quantified equalities, mentioned in section 3.1. We will also present a simple decision procedure for satisfiability checking. To do that we need a rather standard notions of terms, substitutions, unifiers, etc. For the sake of completeness we give these definitions here. We also mention a standard unification algorithm as a decision procedure for equality constraints. This view of unification bridges the gap between the conventional and constraint logic programming.

Let us start with definitions of terms and substitutions.

Definition 1. Given an infinite set of variables V and a finite set of constructor symbols $C = \{C^i_{n_i}\}_i$, each with associated arity n_i, the set of *terms* T is inductively defined as follow:

- $\forall v \in V.\, v \in T$ — every variable is a term;
- $\forall c_k \in C.\, \forall t_1 \dots t_k \in T.\, c_k(t_1, \dots, t_k) \in T$ — every application of k-ary constructor symbol to k terms is a term.

From now on we will assume that terms are untyped (unsorted) and that there exists an infinitely many constructors of any arity.

Definition 2. Two terms t and u are *syntactically equal*, denoted as $t = u$, iff either

- $t = v$ and $s = v$ for some variable v;
- $t = c_k(t_1, \dots, t_k)$, $s = c_k(s_1, \dots, s_k)$ and $\forall i \in \{1..k\}.\, t_i = s_i$.

Definition 3. A substitution σ is a function from variables to terms: $\sigma : V \longrightarrow T$, s.t. $\sigma(x) \neq x$ only for a finite number of variables. Every substitution σ can be represented as a finite list of pairs $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$. By $dom(\sigma)$ we denote the set $\{x_1, \dots, x_n\}$ and by $codom(\sigma)$ we denote the set $\{t_1, \dots, t_n\}$. We denote empty substitution as $\top$. We also extend the set of substitutions defined above with the one additional element $\bot$.

Definition 4. A substitution can be *applied to a term*. The result of an application of σ ($\sigma \neq \bot$) to t, written as $t\sigma$, is a term defined in the following way:

- $x\sigma \triangleq \sigma(x)$;
- $c_k(t_1, \dots, t_k)\sigma \triangleq c_k(t_1\sigma, \dots, t_k\sigma)$.

The result of the application of $\bot$ to any term is undefined.

Lemma 1. Given two substitutions σ and θ, if $\forall v \in V.\, \sigma(v) = \theta(v)$ then $\forall t.\, t\sigma = u\theta$.

Proof. Can be proved by the induction on t. □

We are now ready to define the the satisfiability of *equality constraint.*

Definition 5. Equality constraint $t \equiv u$ is

- *satisfiable* if $\exists\sigma.\, t\sigma = u\sigma$; such σ is called a *unifier* of t and u;
- *unsatisfiable* otherwise.

Next, we show that the standard unification algorithm can be seen as a decision procedure for checking satisfiability of equality constrains. Before that we need to introduce several definitions.

Definition 6. A term t is *subsumed* by the term u, denoted as $t \sqsubseteq u$, iff $\exists \sigma .\, t = u\sigma$. If $t \sqsubseteq u$ we will also say that *t is a more specific* term than u, or *u is a more general* term than t.

Definition 7. A substitution σ is *subsumed* by a substitution τ, denoted as $\sigma \sqsubseteq \tau$, iff $\forall t .\, t\sigma \sqsubseteq t\tau$. If $\sigma \sqsubseteq \tau$ we also say that *σ is a more specific substitution* than τ, or *σ is a more general substitution* than τ.

Definition 8. Given terms t and u their unifier σ is called the *most general unifier*, iff for every other unifier τ σ is more general that τ, $\tau \sqsubseteq \sigma$.

Theorem 1. Given terms t and u they are either not unifiable (meaning that $\forall \sigma .\, t\sigma \neq u\sigma$), or there exists their most general unifier.

Proof. Proof of this statement can be found in [Robinson et al. 1965]. Proof of the termination and correctness of the unification algorithm, used by the most MiniKanren implementations, can be found in [Kumar and Norrish 2010]. □

From now on we will denote the most general unifier of two terms t and u as $mgu(t, u)$. In case of t is not unifiable with u, we assume that $mgu(t, u) = \bot$.

Remark 1. Note we can associate with an equality constraint $t \equiv u$ a logical first-order formula $t = u$. Additionally, we can associate with each substitution a logical first-order formula by the following rules:

- empty substitution $\top$ is associated with the truth constant $\top$;
- $\bot$ is associated with the falsity constant $\bot$;
- $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ is associated with the formula $x_1 = t_1 \wedge \dots \wedge x_n = t_n$.

Now we can have yet another view on unification. We can say that giving the problem of deciding satisfiability of the formula $t = u$, a unification algorithm reduces it to checking satisfiability of a simpler formula, which corresponds to a substitution. Such a formula is either trivially unsatisfiable (in the case of $\bot$) or trivially satisfiable.

Later on we will need the notion of idempotent substitution and idempotent unifier.

Definition 9. Substitution σ is *idempotent* iff $\forall t .\, t\sigma\sigma = t\sigma$

Lemma 2. If two terms are unifiable, there exists their idempotent unifier.

Proof. For the proof of this statement (for the case of unification algorithm, used in MiniKanren), we refer an interested reader to [Kumar and Norrish 2010]. □

We are ready to move on to disequality constraints. We start with the regular (not quantified) disequalities.

Definition 10. A disequality constraint $t \not\equiv u$ is

- *satisfiable* if $\exists \sigma .\, t\sigma \neq u\sigma$;
- *unsatisfiable* otherwise.

Next lemma gives us a simple decision procedure for checking satisfiability of disequalities.

Lemma 3. Disequality constraint $t \not\equiv u$ is

- *satisfiable* if $mgu(t, u) \neq \top$;
- *unsatisfiable* otherwise.

Proof. Let $\theta = mgu(t, u)$. Let us first show that if $\theta = \top$ then disequality is unsatisfiable. By the definition of $\top$ we have $t\theta = t$ and $u\theta = u$, by the definition of unifier $t\theta = u\theta$, and thus $t = u$. From that, it is easy to show that $\forall \sigma .\, t\sigma = s\sigma$, which means that the disequality is unsatisfiable according to the definition 10. If $\theta \neq \top$ then there exists a substitution σ, s.t. $\theta \sqsubseteq \sigma$ (e.g. $\sigma = \top$). Since θ is most general unifier, and σ is more general that θ, then σ is not a unifier, and thus $t\sigma \neq u\sigma$. □

Given lemma 3, the satisfiability of constraint $t \not\equiv u$ can be checked easily. One need to compute $mgu(t, u)$ and if it is not an empty substitution, then constraint is satisfiable.

Remark 2. Given disequality constraint $t \not\equiv u$, a substitution $mgu(t, u)$, can be matched to the logical formula in the following way:

- the empty substitution $\top$ is associated with falsity constant $\bot$;
- $\bot$ is associated with the truth constant $\top$;
- $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ is associated with the formula $x_1 \neq t_1 \vee \dots \vee x_n \neq t_n$.

The following definition introduces universally quantified disequalities.

Definition 11. Universally quantified disequality constraint $\forall \overline{x}.\, t \not\equiv u$ is

- *satisfiable* iff $\exists \sigma.\, \forall \tau, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma \neq u\tau\sigma$
- *unsatisfiable* iff $\forall \sigma.\, \exists \tau, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma = u\tau\sigma$

For the decision procedure of this type of disequalities, we need one auxiliary lemma.

Lemma 4. Given terms t and u consider $\theta = mgu(t, u)$. Assume without the loss of generality that $\overline{x} \not\subseteq codom(\theta)$ (if $v \mapsto x \in \theta$ for some $x \in \overline{x}$ consider $\hat{\theta}$ such that it is equal to θ except that instead of mapping v to x it maps x to v). Universally quantified disequality constraint $\forall \overline{x}.\, t \not\equiv u$ is

- *satisfiable* if $\theta = \bot$ or $dom(\theta) \not\subseteq \overline{x}$
- *unsatisfiable* if $dom(\theta) \subseteq \overline{x}$

Proof. First, let us show that $\theta = \bot$ or $dom(\theta) \not\subseteq \overline{x}$ implies satisfiability. We need to show that $\exists \sigma.\, \forall \tau, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma \neq u\tau\sigma$. Take $\sigma = \top$. Thus $t\tau\sigma = t\tau$ and $u\tau\sigma = u\tau$. It is left to show that $\forall \tau, dom(\tau) \subseteq \overline{x}.\, t\tau \neq u\tau$. If $\theta = \bot$ then t and u are not unifiable, which implies the above statement. Otherwise, consider some τ such that $dom(\tau) \subseteq \overline{x}$. If $t\tau = u\tau$ then τ is a unifier of t and u. Thus $\tau \sqsubseteq \theta$. If we will show that $dom(\theta) \subseteq dom(\tau) \subseteq \overline{x}$ we will get a contradiction with the our assumptions and therefore $t\tau \neq u\tau$. Indeed, consider $v \in dom(\theta)$. If $v \notin dom(\tau)$ consider two cases:

- $v \mapsto w \in \theta$ for some $w \in V$. From the assumptions follows that $w \notin \overline{x}$ and thus $w \notin dom(\tau)$. Consider the term $f(v, w)$ for some binary constructor f. It is easy to see that $f(v, w)\tau = f(v, w) \not\sqsubseteq f(w, w) = f(v, w)\theta$, which contradicts $\tau \sqsubseteq \theta$. Thus it should be that $v \in dom(\tau)$.
- $v \mapsto s \in \theta$ for some term $s \notin V$. Then, trivially $v\tau = v \not\sqsubseteq s = v\theta$, which contradicts $\tau \sqsubseteq \theta$. Thus it should be that $v \in dom(\tau)$.

Finally, let us show that if $dom(\theta) \subseteq \overline{x}$ then $\forall \sigma.\, \exists \tau, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma = u\tau\sigma$. Indeed, consider some σ. Take $\tau = \theta$. Then $t\tau = u\tau$ and thus $t\tau\sigma = u\tau\sigma$. □

By the above lemma, if the substitution $mgu(t, u)$ maps only universally quantified variables, then the disequality is unsatisfiable and satisfiable otherwise.

Finally, it is left to show how to check satisfiability of the quantified equalities of the form $\forall \overline{x}.\, \exists \overline{y}.\, t \equiv u$. As we will see soon, if the constraint of this form is satisfiable, then the logical formula, corresponding to the constraint, $\forall \overline{x}.\, \exists \overline{y}.\, t = u$ is equivalent to the formula $\exists \overline{y^*}.\, \bigwedge_i v_i = t_i$ such that $v_i \in V$ and $v_i \notin \overline{x}$ and $Vars(t_i) \cap \overline{x} = \emptyset$ for all i [Liu et al. 1999].

Definition 12. Quantified equality constraint of the form $\forall \overline{x} \exists \overline{y}.\, t \equiv u$ is

- *satisfiable* iff $\exists \sigma.\, \forall \tau, dom(\tau) \subseteq \overline{x}.\, \exists \phi, dom(\phi) \subseteq \overline{y}.\, t\phi\tau\sigma = u\phi\tau\sigma$
- *unsatisfiable* iff $\forall \sigma.\, \exists \tau, dom(\tau) \subseteq \overline{x}.\, \forall \phi, dom(\phi) \subseteq \overline{y}.\, t\phi\tau\sigma \neq u\phi\tau\sigma$

Lemma 5. If terms t and u are not unifiable then the constraint is unsatisfiable. Otherwise let θ be an idempotent unifier of t and u (Lemma 2 states that if terms are unifiable there exists their idempotent unifier). Let $\theta_y \triangleq \{y \mapsto t \mid y \mapsto t \in \theta \wedge y \in \overline{y}\}$. Let $\hat{\theta} = \{v \mapsto t \mid v \mapsto t \in \theta \wedge v \notin \overline{y}\}$. Then the quantified equality constraint of the form $\forall \overline{x} \exists \overline{y}.\, t \equiv u$ is

- *satisfiable* if $dom(\hat{\theta}) \cap \overline{x} = \emptyset$ and $\forall p \in codom(\hat{\theta}).\, Vars(p) \cap \overline{x} = \emptyset$
- *unsatisfiable* if $dom(\hat{\theta}) \cap \overline{x} \neq \emptyset$ or $\exists p \in codom(\hat{\theta}).\, Vars(p) \cap \overline{x} \neq \emptyset$

Proof. First, it is obvious that if t and u are not unifiable then the constraint is unsatisfiable. Next, let us prove the statement involving satisfiability. We need to show that if the given condition is met, then $\exists \sigma.\, \forall \tau, dom(\tau) \subseteq \overline{x}.\, \exists \phi, dom(\phi) \subseteq \overline{y}.\, t\phi\tau\sigma = u\phi\tau\sigma$. Take $\sigma = \hat{\theta}$. Given an arbitrary τ such that $dom(\tau) \subseteq \overline{x}$ take ϕ to be equal to θ_y. To complete the proof we will need an auxiliary statement.

- $\forall s.\, s\tau\hat{\theta} = s\hat{\theta}\hat{\tau}$ where $\hat{\tau} \triangleq \{x \mapsto t\hat{\theta} \mid x \mapsto t \in \tau\}$.

  Proof. By the Lemma 1 it is sufficient to show that $\forall v \in V.\, v\tau\hat{\theta} = v\hat{\theta}\hat{\tau}$. Consider the cases:
  - $v \in \overline{x}$. Then $v\tau\hat{\theta} = \tau(v)\hat{\theta}$. Since $dom(\hat{\theta}) \cap \overline{x} = \emptyset$, $\hat{\theta}(v) = v$. Then $v\hat{\theta}\hat{\tau} = v\hat{\tau}$ and by the construction $v\hat{\tau} = \tau(v)\hat{\theta}$.
  - $v \notin \overline{x}$. Then $v\tau\hat{\theta} = \hat{\theta}(v)$. Since $\forall p \in codom(\hat{\theta}).\, Vars(p) \cap \overline{x} = \emptyset$, $v\hat{\theta}\hat{\tau} = v\hat{\theta}$ and trivially $v\hat{\theta} = \hat{\theta}(v)$.

  ■

By this statement $t\theta_y\tau\hat{\theta} = t\theta_y\hat{\theta}\hat{\tau}$ and $u\theta_y\tau\hat{\theta} = u\theta_y\hat{\theta}\hat{\tau}$. Because θ is idempotent $t\theta = t\theta_y\hat{\theta}$ and $u\theta = u\theta_y\hat{\theta}$. Finally, since θ is a unifier $t\theta = u\theta$.

It is left to prove the statement involving unsatisfiability. In fact, we will prove more general statement.

- Let $\tilde{t}$ and $\tilde{u}$ be two arbitrary unifiable terms, let $\tilde{\theta}$ be their unifier. Then
  $\forall \overline{x} \subseteq dom(\tilde{\theta}) \cup \bigcup_{p \in codom(\tilde{\theta})} Vars(p), \overline{x} \neq \emptyset.\, \forall \sigma.\, \exists \tau, dom(\tau) \subseteq \overline{x}.\, \tilde{t}\tau\sigma \neq \tilde{u}\tau\sigma$

  Proof. By the induction on t:
  - $\tilde{t} = v$ for some $v \in V$. Consider the cases for u:
    - $\tilde{u} = w$ for some $w \in V$. Then $\tilde{\theta} = \{v \mapsto w\}$ and either $v \in \overline{x}$ or $w \in \overline{x}$.
      Let the former be true (the other case is similar). Given some arbitrary σ
      take $\tau \triangleq \{v \mapsto z \mid z \in V \setminus (dom(\sigma) \cup \bigcup_{p \in codom(\sigma)} Vars(p))\}$. Then $v\tau\sigma = z$ and $w\tau\sigma \neq z$.
    - $\tilde{u} = g(\tilde{u_1}, \ldots, \tilde{u_m})$ for some constructor g. Then $\tilde{\theta} = \{v \mapsto u\}$. If $v \in \overline{x}$ then pick some constructor $f \neq g$ (because we assume there exists an infinite number of constructor symbols, we can always do it). Take $\tau \triangleq \{v \mapsto f(z_1, \ldots, z_n) \mid z_i \in V\}$. Then $v\tau\sigma = f(\tilde{t'_1}, \ldots, \tilde{t'_n})$ and $\tilde{u}\tau\sigma = g(\tilde{u'_1}, \ldots, \tilde{u'_m})$, and thus these terms are not equal. If $v \notin \overline{x}$ then take some $x \in \overline{x}$. For some i it should be that $x \in Vars(\tilde{u_i})$. Given σ consider $\sigma(v)$, pick some s such that $\sigma(v) \neq g(\tilde{u_1}, \ldots, \tilde{u_m})\{x \mapsto s\}$ (it can be done by the induction on $\sigma(v)$). Then $\tau \triangleq \{x \mapsto s\}$.
  - $\tilde{t} = f(\tilde{t_1}, \ldots, \tilde{t_n})$. Consider the cases for u:
    - $\tilde{u} = w$ for some $w \in V$. Then the proof proceeds in the same way as in the previous case.
    - $\tilde{u} = f(\tilde{u_1}, \ldots, \tilde{u_n})$ ($\tilde{u}$ cannot be equal to some constructor $g \neq f$ by our assumption of unifiability of terms). Then by our assumption $\tilde{t_1} \equiv \tilde{u_1} \wedge \cdots \wedge \tilde{t_n} \equiv \tilde{u_n}$. For some i it should be the case that $\overline{x} \subseteq dom(\tilde{\theta_i}) \cup \bigcup_{p \in codom(\tilde{\theta_i})} Vars(p)$ where $\theta_i \triangleq mgu(\tilde{t_i}, \tilde{u_i})$. By the induction for an arbitrary σ there exists τ such that $\tilde{t_i}\tau\sigma \neq \tilde{u_i}\tau\sigma$ and thus $\tilde{t}\tau\sigma \neq \tilde{u}\tau\sigma$.

  ■

□

Given this lemma, we compute $mgu(t, u)$ in order to check the constraint $\forall\overline{x}\exists\overline{y}.\, t \equiv u$. By $mgu(t, u)$ we can construct an idempotent substitution θ that is also a unifier of t and u. We take $\hat{\theta}$ — a part of θ that does not bind existentially quantified variables $\overline{y}$. Then we check if $\hat{\theta}$ binds variables from $\overline{x}$, or some term from codomain of $\hat{\theta}$ contains variables from $\overline{x}$. If it does then the constraint is unsatisfiable, because in this case we can always pick an assigment for $\overline{x}$ that will make the terms not unifiable. Otherwise it is satisfiable.

3.3 Extending the Search

In this section we describe how OCanren search interacts with negation. Also we finally present the code of negation operator itself.

In OCanren (as in any typical MiniKanren implementation) the search is implemented on top of backtracking lazy stream monad [Kiselyov et al. 2005]. During the search the current state is maintained. The state contains accumulated constraints plus some supplementary information stored in the environment (for example, the identifier of last allocated variable). A goal is simply a function which takes a state and returns a lazy stream of states. All logical primitives, such as individual constraints, conjunction, disjunction, and fresh variable introduction, can be implemented based on this representation of goals (an interested reader may refer to [Hemann and Friedman [n. d.]; Kiselyov et al. 2005]).

Now we can define the negation operator ¬ (see Listing 6). Let us describe it in details.

```
 1  let (¬) g st =
 2    let sts′ = g st in
 3    let cexs = Stream.map (diff st) sts′ in
 4    let sub ss cex =
 5      let ss′ = negate cex in
 6      Stream.bind ss  (fun s  →
 7      Stream.bind ss′ (fun s′  →
 8        Stream.unit (merge s s′)
 9      ))
10    in
11    Stream.fold sub (Stream.unit st) cexs
```

Listing 6. Implementation of the negation operator

Negation operator is a function which takes a goal and returns a negated goal. Because a goal is itself a function taking a state, (¬) takes two arguments: the goal g and the state `st` (line 1).

The first step of constructive negation is to run the positive version of the goal, as code in line 2 does. We run it in the current state `st` and thus the call `g st` returns a stream of refined states `sts′`. Each state from this stream will contain the constraints from the original state `st` as its subpart. However, we need to negate only constraints originated from g solely. Thus, on the line 3 we map every state from the stream `sts′` to its *difference* with respect to the original substitution `st`. In order to compute difference of two states `st` and `st′` (Listing 7), given that `st` is more general that `st′` we need to compute difference of their substitutions and disequality constraints stores. The difference of substitution `s′` with respect to `s` (Listing 8) is just a substitution containing all mappings from `s′` which are not simultaneously in `s`. The difference of constraint store `c′` with respect to `c` is a constraint store containing all disequalities that are in `c'` but not in `c` (Listing 8). As long as persistent data structures are used to implement substitutions and constraint stores, the `diff` can be computed[4].

Line 4 defines auxiliary function `sub` which takes two arguments: a stream of states `ss` and some state `cex`, and returns another stream. The purpose of this function it to "subtract" `cex` from every element of `ss`. It is done as follows. First, the state `cex` is negated as described in section 3.1 (line 5). As we have seen, as a result of the negation of a single state the stream of states (the disjunction of formulas) can be obtained. Thus the result of the call `negate cex` is the stream of states `ss'`. For every combination of some state `s` from the given stream `ss` (line 6) and some state `s'` from the stream `ss'` representing the result of negation (line 7) we compute their conjunction (line 8). The conjunction of two states is computed by the function `merge` (Listings 7, 8).

Finally, on the line 11 `fold` is called on the stream `cexs`, which is a stream of answers for the positive version of the goal `g`, with function `sub` defined above and the initial accumulator `Stream.unit st`. The function `Stream.fold` is implemented as a regular left fold over a possibly infinite list. Intuitively with folding over stream `cexs` we "subtract" from the original state `st` every answer obtained from the goal `g`.

[4] In the Listing 8 we present a simple representation of the constraint store as a list of substitutions. In the actual implementation, we use more sophisticated representation, that also provides the `diff` function. The simpler version presented here gives some intuition on how to implement `diff` for the constraint stores.

```
module State = struct
  type t = Env.t * Subst.t * CStore.t

  ...

  let diff (e, s, cs) (e′, s′, cs′) =
    let e  = Env.diff e e′ in
    let s  = Subst.diff s s′ in
    let cs = CStore.diff cs cs′ in
    (e, s, cs)

  let merge (e, s, cs) (e′, s′, cs′) =
    let e = Env.merge e e′ in
    match Subst.merge s s′ with
    | None     →  None
    | Some s   →
      let cs = CStore.merge cs cs′ in
      match CStore.recheck s cs with
      | None     →  None
      | Some cs  →  (e, s, cs)
end
```

Listing 7. Implementation of the auxiliary functions

```
module VarMap = Map.Make(Var)

module Subst = struct
  (* Substitution is a mapping from variables to terms *)
  type t = Term.t VarMap.t
  ...

  let diff s s′ =
    VarMap.fold (fun v t a  →
      if not (VarMap.mem v s) then
        VarMap.add v t a
      else a
    ) s′ VarMap.empty

  let merge s s′ =
    VarMap.fold (fun v t  →  function
      | None    →  None
      | Some a  →  unify a v t
    ) s′ (Some s)
end

module CStore = struct
  (* Constraint store is a list of substitutions *)
  type t = Subst.t list
  ...

  (* cs′ must be obtained from cs by
   * the addition of new constraints
   *)
  let diff cs cs′ =
    if cs′ = cs then []
    else
      match cs′ with
      | _ :: cs′  →  diff cs cs′
      (* cs′ = [] implies cs = []  *)
      | _  →  assert false

  let merge cs cs′ =
    List.append cs′ cs
end
```

Listing 8. Implementation of the auxiliary functions

3.4 Stratification

A negation, when combined with recursion, might become a source of confusion for a programmer. Consider the program in Listing 9. It encodes a two-players game. The positions in the game are given as single-character strings `'a'`, `'b'`, `'c'` and `'d'`. A binary relation `move` encodes the game field as the set of possible moves. An unary relation `winning` determines the set of winning positions, meaning that if the first player starts from some winning position, by making "good" moves the player has an opportunity to win the game. According to the definition of `winning`, the position is winning if there exists a move from this position to some non-winning position. Clearly, every position with no moves from it is losing.

Given the suchlike definition of `winning`, there is no doubt, that the position `'d'` is losing position, and thus the goal `winning 'd'` should fail, which, in turn, means that `winning 'c'` should succeed. However, whether the goal `winning 'a'` (or `winning 'b'`) should fail or succeed is not clear.

```
let move x y =
  (x, y) ≡ ('a', 'b') ∨
  (x, y) ≡ ('b', 'a') ∨
  (x, y) ≡ ('b', 'c') ∨
  (x, y) ≡ ('c', 'd')

let winning x =
  fresh (y) (
    (move x y) ∧ ¬(winning y)
  )
```

Listing 9. Encoding of two-players game

The problem with the semantics of program in Listing 9, originates from the interaction of negation and recursion. Definition of the relation `winning` refers to itself under negation. Logic programs that have this property are called *non-stratified* [Przymusinski 1989]. Vice versa, programs that do not have loops over negation, are called *stratified*.

Our current implementation handles only stratified programs. We leave the task of supporting non-stratified programs as a direction for future work.

4 EVALUATION

In this section, we present an evaluation of implemented constructive negation on a series of examples.

4.1 If-then-else

Using relational if-then-else operator, presented in section 2.1, we have implemented several higher-order relations over lists, namely `find` (Listing 10), `remove`[5] (Listing 11) and `filter` (Listing 12). These relations are almost identical (syntactically) to their functional implementations. We have tested that these relations can be run in various directions and produce the expected results. For example, the goal `filter p q q` with the predicate `p` equal to

```
fun l → fresh (x) (l ≡ [x])
```

stating that the given list should be a singleton list, starts to generate all singleton lists. Vice versa, the goal `filter p q []` with that same `p` generates all lists, constrained to be not a singleton list.

[5]Note, this implementation differs from the one in Section 1, but it is easy to see that these two are semantically equivalent.

Listings 13-16 give more concrete examples of queries to these relations. In the listing the syntax run n q g means running a goal g with the free variable q taking the first n answers ("*" denotes all answers). After the sign ⇝ the result of the query is given. The result fail means that the query has failed. The result succ $\{\{a_1\}; \ldots \{a_n\}\}$ means that the query has succeeded delivering n answers. Each answer represents a set of constraint on free variables. Constraints are of two forms: equality constraints, e.g. q = (1, $\_ .{}_0$), or disequality constraints, e.g. q ≠ (1, $\_ .{}_0$). The terms of the form $\_ .{}_i$ in the answer denote some universally quantified variables.

```
let find p e xs =
  fresh (x xs′ ys′) (
    xs ≡ x::xs′ ∧
    ifte (p x)
      (e ≡ x)
      (find p e xs′)
  )
```

Listing 10. A definition of find relation

```
let remove p xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs′ ys′) (
    xs ≡ x::xs′ ∧
    ifte (p x)
      (ys ≡ xs′)
      (ys ≡ x::ys′ ∧
       remove p xs′ ys′)
  )
```

Listing 11. A definition of remove relation

```
let filter p xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs′ ys′) (
    xs ≡ x::xs′ ∧
    (ifte (p x)
      (ys ≡ x :: ys′)
      (ys ≡ ys′)) ∧
    filter p xs′ ys′
  )
```

Listing 12. A definition of filter relation

```
let p l = fresh (x) (l ≡ [x])
```

Listing 13. Definition of the predicate p

```
run 3 q (fresh (e) find p e q)
⇝ succ {
     { q = [_.0] :: _.1 }
     { q = _.0 :: [_.1] :: _.2;
          _.0 ≠ [_.3] }
     { q = _.0 :: _.1 :: [_.2] :: _.3;
          _.0 ≠ [_.4]; _.1 ≠ [_.5] }
   }
```

Listing 14. Example of queries to find

```
run * q (fresh (e) remove p q [[ ]])
⇝ succ {
     { q = [[_.0]; [ ]] }
     { q = [[ ]] }
     { q = [[ ]; [_.0]] }
   }

run 3 q (fresh (e) remove p q q)
⇝ succ {
     { q = [] }
     { q = [_.0], _.0 ≠ [_.1] }
     { q = [_.0; _.1];
          _.0 ≠ [_.2]; _.1 ≠ [_.3] }
   }
```

Listing 15. Example of queries to remove

```
run 3 q (filter p q q)
⇝ succ {
     { q = [ ] }
     { q = [_.0] }
     { q = [_.0; _.1] }
   }

run 3 q (filter p q [1])
⇝ succ {
     { q = [[1]] }
     { q = [_.0; [1]]; _.0 ≠ [_.1] }
     { q = [[1]; _.0]; _.0 ≠ [_.1] }
   }

run 3 q (filter p q [ ])
⇝ succ {
     { q = [] }
     { q = [_.0]; _.0 ≠ [_.1] }
     { q = [_.0; _.1];
             _.0 ≠ [_.2]; _.1 ≠ [_.3] }
   }
```

Listing 16. Example of queries to `filter`

4.2 Universal quantification

In the Section 2.2 we presented the `forall` goal constructor which is implemented through the double negation. We have observed, that although `forall g` does not terminate when the goal `g x` has an infinite number of answers (assuming `x` is a fresh variable), it does terminate in the case when `g x` has a finite number of answers. The behavior of `forall` in this case is sound even in the presence of disequality constraints or nested quantifiers.

The Table 1 gives some concrete examples. The left column contains the tested goals[6] and the right column gives the obtained results. For the results we use the same notation as in the previous section.

5 LIMITATIONS AND FUTURE WORK

In this section we discuss the limitations of constructive negation in general and our implementation in particular. Also we consider possible directions for future work.

5.1 Type Constraints

Although the program written in OCanren typechecks statically (thus, for example, preventing the user from unifying two terms of distinct types), at runtime the type information is erased. In the presence of even regular disequality constraints it can lead to the incorrect results. As an example, consider the following program:

[6] We typeset the goals in terms of first-order logic syntax instead of OCanren syntax for brevity and clarity.

$\forall x.\, x = q$	`fail`
$\forall x.\, \exists y.\, x = y$	`succ {[q = _.0]}`
$\forall x.\, \exists y.\, x = y \wedge y = q$	`fail`
$\forall x.\, q = (1, x)$	`fail`
$\forall x.\, \exists y.\, y = (1, x)$	`succ {[q = _.0]}`
$\forall x.\, \exists y.\, x = (1, y)$	`fail`
$\forall x.\, x \neq q$	`fail`
$\forall x.\, \exists y.\, x \neq y$	`succ {[q = _.0]}`
$\forall x.\, \exists y.\, x \neq y \wedge y = q$	`fail`
$\forall x.\, q \neq (1, x)$	`succ {[q ≠ (1, _.0)]}`
$(\exists x.\, q = (1, x)) \wedge (\forall x.\, q \neq (1, x))$	`fail`
$\forall x.\, (x, x) \neq (0, 1)$	`succ {[q = _.0]}`
$\forall x.\, (x, x) \neq (1, 1)$	`fail`
$\forall x.\, (x, x) \neq (q, 1)$	`succ {[q ≠ 1]}`
$\exists a\ b.\, q = (a, b) \wedge \forall x.\, (x, x) \neq (a, b)$	`succ {[q = (_.0, _.1); _.0 ≠ _.1]}`

Table 1. `forall` evaluation

```
type bool = true | false

let g =
  fresh (x y z : bool) (
    (x ≢ y)
    (y ≢ z)
    (z ≢ x)
  )
```

The goal `g` states that there exists at least three different non-equal terms of type `bool`, which, as we know, is not true. Yet the query `run g` will succeed.

In order to prevent unsoundness in cases like this, type information in the form of *type constraints* should be somehow attached to the variables at runtime. The satisfiability of type constraints then should be rechecked each time when the new disequality is added to some variable. An extension of OCanren with type constraints is a direction for future work.

5.2 Non-stratified Programs

As we have already discussed in the section 3.4 our current implementation handles only stratified logic programs. One of the possible extensions is to support non-stratified programs, such as one given in Listing 9, with respect to well-founded and/or stable model semantics (see section 6 for the details).

5.3 Negation of Goals With an Infinite Number of Answers

Consider the following program:

```
let zeros l =
  l ≡ [0]
  ∨
  fresh (l′) (
    (l ≡ 0 :: l′)
    (zeros l′)
  )
```

The unary relation `zeros` defines lists consisting of zeros. Now, intuitively, the query `run ¬(zeros q)` should enumerate all lists that are not built out of zeros only. Yet this query will fail to deliver even a single answer. Why? Consider its operational behavior. First the positive version of the goal, that is `zeros q`, should be executed. Then all answers to this goal should be collected and complemented. However, there is an infinite number of answers to `zeros q` and thus this process will never terminate.

It is a significant drawback of constructive negation that the negation of the goal cannot be computed if the goal has an infinite number of answers. This limitation cannot be avoided in general, however in some cases it is possible to narrow the number of answers to some subgoal by the reordering of surrounding subgoals. For example, the query `run ¬(zeros q) ∧ (q ≡` [1]) can be executed in finite time by the reordering of conjuncts. It seems that the best strategy is to delay negative subgoals as long as possible, but we do not have a formal proof of that.

6 RELATED WORKS

There are two directions of work in the process of incorporating negative reasoning in the logic programming: the first considers the semantics of negation, and the second is focused mainly on implementation aspects.

The first attempt to give a semantics for negation in logic programming was done by Clark [Chan 1988; Clark 1978] with his completion semantics. It was then realized, that Clark's semantics has various drawbacks [Van Gelder et al. 1991].

Przymusinski [Przymusinski 1989] has studied the semantics of stratified logic programs. He introduced the notion of *perfect model semantics* for such programs. Stratified logic programs have a variety of good properties, including the property that each stratified program has a unique minimal model.

In an attempt to extend the semantics of negation to non-stratified programs the *well-founded semantics* was proposed [Van Gelder et al. 1991]. However, this semantics is three-valued, meaning that for some queries it can return answer `unknown`. For example, given the relation `winning` (section 3.4, listing 9), queries `winning 'a'` and `winning 'b'` would return `unknown`.

An alternative approach is *stable model semantics* [Gelfond and Lifschitz 1988]. Under this semantics, non-stratified logic program can have several stable models. Program, that defines `winning`, has two stable models, in one of these models goal `winning 'a'` succeeds and `winning 'b'` fails, in the other `winning 'a'` fails and `winning 'b'` succeeds. Logic programming under stable model semantics is also known under the name answer set programming (ASP).

The works [Dovier et al. 2000; Stuckey 1991] are theoretical studies of constructive negation in the context of constraint logic programming. They give a necessary and sufficient condition for the constraint structures that are compatible with constructive negation. Namely, the constraint structure should be *admissible closed*.

From an implementation side, Chen et al. [Chen et al. 1995; Chen and Warren 1996] developed a Prolog system based on SLG resolution, which is sound with respect to well-founded semantics. However, they used negation as failure with delaying of non-ground negative subgoals. [Liu et al. 1999] is an extension of this system with the support of the constructive negation. Works [Álvez et al. 2004; Barták 1998] implement a constraint logic programming systems with the support of constructive negation. Yet, as with our implementation, the constructive negation in these systems supports only equality and disequality constraints over first-order terms. We are not aware of any practical implementation that is parametric over arbitrary admissible closed constraint structures.

Many tools were developed to compute stable models of logic programs, among them are [Gebser et al. 2007; Giunchiglia et al. 2006]. These systems usually require to perform grounding of logic program. The problem of finding stable models of ground logic program then is encoded as propositional formula and solved by some SAT solver. Unfortunately, some logic programs do not have finite grounding, but even if a program has it, grounding may cause an exponential blow-up. Recently, a goal-directed system for computing stable models was developed [Arias et al. 2018; Marple et al. 2012, 2017]. To the best of our knowledge, it is the only ASP system, that does not require grounding. The key components of this system are the usage of tabling, constructive negation, coinductive logic programming, and non-monotonic reasoning check. It is an interesting and challenging task to extend MiniKanren with the support of stable model semantics in the spirit of this line of work.

7 CONCLUSION

We have presented an implementation of constructive negation for relational programming language OCanren, a dialect of MiniKanren. Unlike the negation as failure, constructive negation is consistent with the pure logical nature of MiniKanren. As we have demonstrated the negative reasoning increases the expressive power of relational language by allowing to compose more relations in a natural and intuitive form.

ACKNOWLEDGMENTS

We want to thank Dmitry Boulytchev, Ekaterina Verbitskaia and anonymous reviewers for valuable comments on a draft version of the paper. This work was partially supported by the grant 18-01-00380 from the Russian Foundation for Basic Research and the grant from JetBrains Research.

REFERENCES

Javier Álvez, Paqui Lucio, and Fernando Orejas. 2004. Constructive negation by bottom-up computation of literal answers. In *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 1468–1475.

Joaquin Arias, Manuel Carro, Elmer Salazar, Kyle Marple, and Gopal Gupta. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 337–354.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

Roman Barták. 1998. Constructive negation in clp (h). *submitted to CP* 98 (1998).

William E Byrd. 2010. Relational programming in miniKanren: techniques, applications, and implementations. (2010).

David Chan. 1988. Constructive negation based on the completed database. In *Proc. of ICLP-88*.

Weidong Chen, Terrance Swift, and David S Warren. 1995. Efficient top-down computation of queries under the well-founded semantics. *The Journal of logic programming* 24, 3 (1995), 161–199.

Weidong Chen and David S Warren. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)* 43, 1 (1996), 20–74.

Keith L Clark. 1978. Negation as failure. In *Logic and data bases*. Springer, 293–322.

Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. 2000. A necessary condition for constructive negation in constraint logic programming. *Inform. Process. Lett.* 74, 3-4 (2000), 147–156.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The MIT Press.

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. clasp: A conflict-driven answer set solver. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 260–265.

Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming.. In *ICLP/SLP*, Vol. 88. 1070–1080.

Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning* 36, 4 (2006), 345.

Jason Hemann and Dan Friedman. [n. d.]. μkanren: A minimal functional core for relational programming, November 2013. *URL http://www.schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf* ([n. d.]).

Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.

Dmitry Kosarev and Dmitry Boulytchev. 2018. Typed embedding of a relational language in OCaml. *arXiv preprint arXiv:1805.11006* (2018).

Ramana Kumar and Michael Norrish. 2010. (Nominal) unification by recursive descent with triangular substitutions. In *International Conference on Interactive Theorem Proving*. Springer, 51–66.

Julie Yuchih Liu, Leroy Adams, and Weidong Chen. 1999. Constructive negation under the well-founded semantics. *The Journal of Logic Programming* 38, 3 (1999), 295–330.

Kyle Marple, Ajay Bansal, Richard Min, and Gopal Gupta. 2012. Goal-directed execution of answer set programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*. ACM, 35–44.

Kyle Marple, Elmer Salazar, Zhuo Chen, and Gopal Gupta. 2017. The s (ASP) Predicate Answer Set Programming System. *The Association for Logic Programming Newsletter* (2017).

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

Lee Naish. 1995. Pruning in logic programming. In *University of Melbourne*. Citeseer.

Teodor C Przymusinski. 1989. *On constructive negation in logic programming*. MIT Press Cambridge, Massachusetts.

John Alan Robinson et al. 1965. A machine-oriented logic based on the resolution principle. *J. ACM* 12, 1 (1965), 23–41.

Peter J Stuckey. 1991. Constructive negation for constraint logic programming. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 328–339.

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.