Certified Semantics for Disequality[*]

DMITRY ROZPLOKHAS, Higher School of Economics, JetBrains Research, Russia
DMITRY BOULYTCHEV, Saint Petersburg State University, JetBrains Research, Russia

We present an extension of our prior work on certified semantics for core miniKanren, introducing disequality constraints in the language. Semantics is parameterized by an exact definition of constraint stores, allowing us to cover different implementations, and we provide a list of sufficient conditions on this definition for search completeness. We also give two examples of concrete implementations of constraint stores that satisfy those sufficient conditions. The description and proofs for parameterized semantics and both implementations are certified in Coq and two correct-by-construction interpreters are extracted.

1 INTRODUCTION

In its initial form [Friedman et al. 2005; Hemann and Friedman 2013] miniKanren introduces a single form of constraint — unification of terms. While from a theoretical standpoint unification together with other primitive constructs (conjunction, disjunction, and fresh variable introduction) form a Turing-complete basis, in practice of relational programming a number of extensions are often used to make specifications more expressive, concise or efficient. One of the most important extensions is *disequality constraint*.

A generic concept of domain-specific constraints in logic programming is studied in details in [Jaffar et al. 1998]; more specifically, disequality constraint [Comon-Lundh 1991] introduces one additional type of base goal — a disequality of two terms

$$t_1 \not\equiv t_2$$

The informal semantics of disequality constraint is complementary to that of unification: it puts certain restrictions on free variables in the terms which prevent them from turning into syntactically equal. Similarly to unification, whose evaluation results in a substitution, which is then threaded through the rest of computations, the effect of disequality constraint is recorded in a *constraint store* which is later used to check the violation of disequalities [Alvis et al. 2011].

We present an extension of our prior work on certified semantics for core miniKanren [Rozplokhas et al. 2019]. In that work, we defined denotational and operational semantics and proved the soundness and completeness of the latter w.r.t. the former. The main advantage of the operational semantics introduced there over the ones developed before [Kumar 2010; Lozov et al. 2017; Rozplokhas and Boulytchev 2018] was its ability to capture the conventional for miniKanren *interleaving search* [Kiselyov et al. 2005] procedure. This allowed us to give the first to our knowledge formal proof of completeness of the interleaving search as the capability to reach all the solutions from denotational semantics (proof of completeness as the fairness of steams interleaving for a

[*]The reported study was funded by RFBR, project number 18-01-00380

Authors' addresses: Dmitry Rozplokhas, Higher School of Economics, JetBrains Research, Russia, rozplokhas@gmail.com; Dmitry Boulytchev, Saint Petersburg State University, JetBrains Research, Russia, dboulytchev@math.spbu.ru.

miniKanren.org/workshop/2021/8-ART6

specific implementation was given in [Hemann et al. 2016]). The development was formally certified in Coq proof assistant [Bertot and Castéran 2004], and a correct-by-construction interpreter was extracted.

To some extent our work follows the conventional roadmap of adding constraints to a pure logic/relational language [Jaffar et al. 1998]; the difference is that, first, we use a specific constraint and a concrete solver, and second, we prove all the results with regard to conventional for miniKanren interleaving search (versus a very generic and abstract breadth-first search in [Jaffar et al. 1998]).

The contribution of our current work is as follows:

- we extend our denotational semantics to handle disequality constraints;
- we introduce a new abstraction layer (a constraint store with a number of abstract operations) in our operational semantics;
- we formulate a set of *sufficient conditions for completeness*, expressed as algebraic properties of constraint store and abstract operators, and prove the soundness and completeness of the extended operational semantics w.r.t. the denotational one;
- we present two concrete implementations of constraint store and abstract operators and show that they satisfy the sufficient conditions; thus, the soundness and completeness of the implementation with disequality constraints follow immediately, and correct-by-construction interpreter for miniKanren with disequality constraints can be extracted
- we demonstrate how our framework can be used to prove some properties of implementations of disequality constraints.

The paper is organized as follows. In Section 2 we recall our framework from the previous paper [Rozplokhas et al. 2019], which is extended in this work. The following sections describe the new results. Section 3 contains the description of the extensions in semantics and sufficient conditions on abstract definitions for search completeness. Section 4 contains two examples of implementations of constraint stores that satisfy the sufficient conditions for completeness. Section 5 presents some applications of the extended semantics. The final section concludes.

2 THE SYNTAX AND SEMANTICS OF THE CORE LANGUAGE

In this section, we recall existing definitions of the syntax and the two semantics for the core language without disequality constraints and the main result — the equivalence of these two semantics [Rozplokhas et al. 2019].

2.1 The Syntax of Core Language

The syntax of the language is shown in Fig. 1. First, we fix a set of constructors C with known arities and consider a set of terms $\mathcal{T}_X$ with constructors as functional symbols and variables from X. We parameterize this set with an alphabet of variables since in the semantic description we will need *two* kinds of variables. The first kind, *syntactic* variables, is denoted by $\mathcal{X}$. We also consider an alphabet of *relational symbols* $\mathcal{R}$ which are used to name relational definitions. The central syntactic category in the language is a *goal*. In our case, there are five types of goals: *equality* of terms, conjunction and disjunction of goals, fresh variable introduction, and invocation of some relational definition. Thus, equality is used as a constraint, and multiple constraints can be combined using conjunction, disjunction, and recursion. For the sake of brevity we abbreviate immediately nested "**fresh**" constructs into the one, writing "$\textbf{fresh}\ \ x\, y\, \ldots .\ g$" instead of "$\textbf{fresh}\ \ x\ .\ \ \textbf{fresh}\ \ y\ .\ \ \ldots g$". The final syntactic category is *specification* $\mathcal{S}$. It consists of a set of relational definitions and a top-level goal. A top-level goal represents a search procedure which returns a stream of substitutions for the free variables of the goal. The language we defined is first-order, as goals can not be passed as parameters, returned or constructed at runtime.

As an example consider the specification for the standard append^o relation and a query which splits a list containing three constants A, B and C into two parts in every possible way:

$\mathcal{C}$	=	$\{C_i^{k_i}\}$	constructors with arities
$\mathcal{T}_X$	=	$X \cup \{C_i^{k_i}(t_1, \dots, t_{k_i}) \mid t_j \in \mathcal{T}_X\}$	terms over the set of variables X
$\mathcal{D}$	=	$\mathcal{T}_\varnothing$	ground terms
$\mathcal{X}$	=	$\{x, y, z, \dots\}$	syntactic variables
$\mathcal{A}$	=	$\{\alpha, \beta, \gamma, \dots\}$	semantic variables
$\mathcal{R}$	=	$\{R_i^{k_i}\}$	relational symbols with arities
$\mathcal{G}$	=	$\mathcal{T}_\mathcal{X} \equiv \mathcal{T}_\mathcal{X}$	equality
		$\mathcal{G} \wedge \mathcal{G}$	conjunction
		$\mathcal{G} \vee \mathcal{G}$	disjunction
		$\textbf{fresh}\ \mathcal{X}\ .\ \mathcal{G}$	fresh variable introduction
		$R_i^{k_i}(t_1, \dots, t_{k_i}),\ t_j \in \mathcal{T}_\mathcal{X}$	relational symbol invocation
$\mathcal{S}$	=	$\{R_i^{k_i} = \lambda\ x_1^i \dots x_{k_i}^i\ .\ g_i;\}\ g$	specification

Fig. 1. The syntax of core language

```
appendᵒ =λ x y xy .
  ((x ≡ Nil) ∧ (xy ≡ y)) ∨
  (fresh h t ty .
     (x  ≡ Cons (h, t))  ∧
     (xy ≡ Cons (h, ty)) ∧
     (appendᵒ t y ty)
  );
appendᵒ x y (Cons (A, Cons (B, Cons (C, Nil))))
```

2.2 Denotational sematics

For denotational semantics, we use a simple set-theoretic approach which can be considered analogous to the least Herbrand model for definite logic programs [Lloyd 1984].

Intuitively, the mathematical model for every goal should be a relation between semantic variables that occur free in this goal. We represent this relation as a set of total functions

$$\mathfrak{f} : \mathcal{A} \mapsto \mathcal{D}$$

from semantic variables to ground terms. We call these functions *representing functions.*

Then, the semantic function for goals is parameterized over environments which prescribe semantic functions to relational symbols:

$$\Gamma : \mathcal{R} \to ({\mathcal{T}_\mathcal{A}}^* \to 2^{\mathcal{A} \to \mathcal{D}})$$

An environment associates with relational symbol a function that takes a string of terms (the arguments of the relation) and returns a set of representing functions. The signature for semantic brackets for goals is as follows:

$$[\![\bullet]\!]_\Gamma : \mathcal{G} \to 2^{\mathcal{A} \to \mathcal{D}}$$

It maps a goal into the set of representing functions w.r.t. an environment Γ.

$$\begin{array}{rcll}
[\![t_1 \equiv t_2]\!]_\Gamma & = & \{\mathfrak{f} : \mathcal{A} \to \mathcal{D} \mid \overline{\mathfrak{f}}\,(t_1) = \overline{\mathfrak{f}}\,(t_2)\} & [\textsc{Unify}_D] \\
[\![g_1 \wedge g_2]\!]_\Gamma & = & [\![g_1]\!]_\Gamma \cap [\![g_1]\!]_\Gamma & [\textsc{Conj}_D] \\
[\![g_1 \vee g_2]\!]_\Gamma & = & [\![g_1]\!]_\Gamma \cup [\![g_1]\!]_\Gamma & [\textsc{Disj}_D] \\
[\![\mathbf{fresh}\, x\,.\, g]\!]_\Gamma & = & ([\![g\,[\alpha/x]]\!]_\Gamma) \uparrow \alpha,\;\; \alpha \notin FV(g) & [\textsc{Fresh}_D] \\
[\![R\,(t_1,\dots,t_k)]\!]_\Gamma & = & (\Gamma\, R)\, t_1 \dots t_k & [\textsc{Invoke}_D]
\end{array}$$

Fig. 2. Denotational semantics of goals

We formulate the following important *completeness condition* for the semantics of a goal g: for any goal g and two representing functions $\mathfrak{f}$ and $\mathfrak{f}'$, such that $\mathfrak{f}|_{FV(g)} = \mathfrak{f}'|_{FV(g)}$

$$\mathfrak{f} \in [\![g]\!] \Leftrightarrow \mathfrak{f}' \in [\![g]\!]$$

In other words, representing functions for a goal g restrict only the values of free variables of g and do not introduce any "hidden" correlations. This condition guarantees that our semantics is complete in the sense that it does not introduce artificial restrictions for the relation it defines. We proved that the semantics of goals always satisfy this condition.

To define the semantic function we need a few operations for representing functions:

- A homomorphic extension of a representing function

$$\overline{\mathfrak{f}} : \mathcal{T}_{\mathcal{A}} \to \mathcal{D}$$

which maps terms to terms:

$$\begin{array}{rcl}
\overline{\mathfrak{f}}\,(\alpha) & = & \mathfrak{f}\,(\alpha) \\
\overline{\mathfrak{f}}\,(C_i^{k_i}\,(t_1,\dots.t_{k_i})) & = & C_i^{k_i}\,(\overline{\mathfrak{f}}\,(t_1),\dots\overline{\mathfrak{f}}\,(t_{k_i}))
\end{array}$$

- A pointwise modification of a function

$$f\,[x \leftarrow v]\,(z) = \left\{\begin{array}{rcl} f\,(z) & , & z \neq x \\ v & , & z = x \end{array}\right.$$

- A *generalization* operation:

$$\mathfrak{f} \uparrow \alpha = \{\mathfrak{f}\,[\alpha \leftarrow d] \mid d \in \mathcal{D}\}$$

Informally, this operation generalizes a representing function into a set of representing functions in such a way that the values of these functions for a given variable cover the whole $\mathcal{D}$. We extend the generalization operation for sets of representing functions $\mathfrak{F} \subseteq \mathcal{A} \to \mathcal{D}$:

$$\mathfrak{F} \uparrow \alpha = \bigcup_{\mathfrak{f} \in \mathfrak{F}} (\mathfrak{f} \uparrow \alpha)$$

The semantics for goals is shown on Fig. 2.

The final component is the semantics of specifications. Given a specification

$$\{R_i = \lambda\, x_1^i \dots x_{k_i}^i\,.\, g_i;\}_{i=1}^n\; g$$

we construct a correct environment Γ_0 and then take the semantics of the top-level goal:

$$[\![\{R_i = \lambda\, x_1^i \dots x_{k_i}^i\,.\, g_i;\}_{i=1}^n\; g]\!] = [\![g]\!]_{\Gamma_0}$$

As the set of definitions can be mutually recursive we apply the fixed point approach and define Γ_0 as the least fixed point of a specific function F that takes an environment Γ and returns new environment in which semantics of a body of each definition is evaluated with environment Γ.

2.3 Operational sematics

The operational semantics of miniKanren, which we described, corresponds to the known implementations with interleaving search. The semantics is given in the form of a labeled transition system (LTS) [Keller 1976].

The states in the transition system have the following shape:

$$S = \mathcal{G} \times \Sigma \times \mathbb{N} \mid S \oplus S \mid S \otimes \mathcal{G}$$

A state has a tree-like structure with intermediate nodes corresponding to partially-evaluated conjunctions ("$\otimes$") or disjunctions ("$\oplus$"). A leaf in the form $\langle g, \sigma, n\rangle$ determines a goal in a context, where g — a goal, σ — a substitution accumulated so far, and n — a natural number, which corresponds to a number of semantic variables used to this point. For a conjunction node, its right child is always a goal since it cannot be evaluated unless some result is provided by the left conjunct.

We also need extended states

$$\overline{S} = \diamond \mid S$$

where $\diamond$ symbolizes the end of the evaluation.

The set of labels is defined as follows:

$$L = \circ \mid \Sigma \times \mathbb{N}$$

The label "$\circ$" is used to mark those steps which do not provide an answer; otherwise, a transition is labeled by a pair of a substitution and a number of allocated variables. The substitution is one of the answers, and the number is threaded through the derivation to keep track of the allocated variables.

The transition rules are shown in Fig. 3. The introduced transition system is completely deterministic.

A derivation sequence for a certain state s determines a *trace* $\mathcal{T}r_s$ — a finite or infinite sequence of answers. The trace corresponds to the stream of answers in the reference miniKanren implementations.

2.4 Semantics Equivalence

After we defined two different kinds of semantics for miniKanren we related them and showed that the results given by these two semantics are the same for any specification. By proving this equivalence we established the *completeness* of the search which means that the search will get all answers satisfying the described specification and only those.

To do it we had to relate the answers produced by these two semantics as they have different forms: a trace of substitutions (along with numbers of allocated variables) for operational and a set of representing functions for denotational. There is a natural way to extend any substitution to a representing function: composing it with an arbitrary representing function will preserve all variable dependencies in the substitution. So we defined a set of representing functions corresponding to substitution as follows:

$$[\![\sigma]\!] = \{\bar{\mathfrak{f}} \circ \sigma \mid \mathfrak{f} : \mathcal{A} \mapsto \mathcal{D}\}$$

And *denotational analog* of an operational semantics (a set of representing functions corresponding to answers in the trace) for given extended state s is then defined as a union of sets for all substitution in the trace:

$$[\![s]\!]_{op} = \cup_{(\sigma,n)\in\mathcal{T}r_s} [\![\sigma]\!]$$

$$\langle t_1 \equiv t_2, \sigma, n\rangle \xrightarrow{\circ} \Diamond,\ \nexists\, mgu\,(t_1\sigma, t_2\sigma) \qquad [\textsc{UnifyFail}]$$

$$\langle t_1 \equiv t_2, \sigma, n\rangle \xrightarrow{(mgu\,(t_1\sigma, t_2\sigma)\circ\sigma),\ n)} \Diamond \qquad [\textsc{UnifySuccess}]$$

$$\langle g_1 \vee g_2, \sigma, n\rangle \xrightarrow{\circ} \langle g_1, \sigma, n\rangle \oplus \langle g_2, \sigma, n\rangle \qquad [\textsc{Disj}]$$

$$\langle g_1 \wedge g_2, \sigma, n\rangle \xrightarrow{\circ} \langle g_1, \sigma, n\rangle \otimes g_2 \qquad [\textsc{Conj}]$$

$$\langle \mathbf{fresh}\ x\,.\,g, \sigma, n\rangle \xrightarrow{\circ} \langle g\,[{}^{\alpha_{n+1}}/_{x}], \sigma, n+1\rangle \qquad [\textsc{Fresh}]$$

$$\frac{R_i^{k_i} = \lambda\, x_1 \dots x_{k_i}\,.\,g}{\left\langle R_i^{k_i}\,(t_1, \dots, t_{k_i}), \sigma, n\right\rangle \xrightarrow{\circ} \left\langle g\,[{}^{t_1}/_{x_1} \dots {}^{t_{k_i}}/_{x_{k_i}}], \sigma, n\right\rangle} \qquad [\textsc{Invoke}]$$

$$\frac{s_1 \xrightarrow{\circ} \Diamond}{(s_1 \oplus s_2) \xrightarrow{\circ} s_2} \qquad [\textsc{DisjStop}]$$

$$\frac{s_1 \xrightarrow{r} \Diamond}{(s_1 \oplus s_2) \xrightarrow{r} s_2} \qquad [\textsc{DisjStopAns}]$$

$$\frac{s \xrightarrow{\circ} \Diamond}{(s \otimes g) \xrightarrow{\circ} \Diamond} \qquad [\textsc{ConjStop}]$$

$$\frac{s \xrightarrow{(\sigma, n)} \Diamond}{(s \otimes g) \xrightarrow{\circ} \langle g, \sigma, n\rangle} \qquad [\textsc{ConjStopAns}]$$

$$\frac{s_1 \xrightarrow{\circ} s_1'}{(s_1 \oplus s_2) \xrightarrow{\circ} (s_2 \oplus s_1')} \qquad [\textsc{DisjStep}]$$

$$\frac{s_1 \xrightarrow{r} s_1'}{(s_1 \oplus s_2) \xrightarrow{r} (s_2 \oplus s_1')} \qquad [\textsc{DisjStepAns}]$$

$$\frac{s \xrightarrow{\circ} s'}{(s \otimes g) \xrightarrow{\circ} (s' \otimes g)} \qquad [\textsc{ConjStep}]$$

$$\frac{s \xrightarrow{(\sigma, n)} s'}{(s \otimes g) \xrightarrow{\circ} (\langle g, \sigma, n\rangle \oplus (s' \otimes g))} \qquad [\textsc{ConjStepAns}]$$

Fig. 3. Operational semantics of interleaving search

This allowed us to state the theorem relating two semantics.

Theorem 1 (Operational semantics soundness and completeness). *For any specification $\{\dots\}\ g$, for which the indices of all free variables in g are limited by some number n*

$$[\![\langle g, \epsilon, n\rangle]\!]_{op} =_n [\![\{\dots\}\ g]\!].$$

Where '$=_n$' means that we compare representing functions of these sets only on the semantic variables from $\{\alpha_1, \dots, \alpha_n\}$:

$$S_1 =_n S_2 \overset{def}{\iff} \{\mathfrak{f}|_{\{\alpha_1, \dots, \alpha_n\}} \mid \mathfrak{f} \in S_1\} = \{\mathfrak{f}|_{\{\alpha_1, \dots, \alpha_n\}} \mid \mathfrak{f} \in S_2\}.$$

We can not use the usual equality of sets instead of this one, the sets from the theorem statement are actually not equal. The reason for this is that denotational semantics encodes only dependencies between the free variables of a goal, which is reflected by the completeness condition, while operational semantics may also contain dependencies between semantic variables allocated in "**fresh**". Therefore we have to restrict representing functions on the semantic variables allocated in the beginning (which includes all free variables of a goal). This does not compromise our promise to prove the completeness of the search as miniKanren provides the result as substitutions only for queried variables, which are allocated in the beginning.

The proof of this main theorem was certified in Coq.

3 EXTENSION WITH DISEQUALITY CONSTRAINTS

In this section, we present extensions of our two semantics for the language with disequality constraints and revised versions of the soundness and completeness theorems.

Disequality constraint introduces one additional type of base goal — a disequality of two terms: $t_1 \not\equiv t_2$

The extension of denotational semantics is straightforward (as disequality constraint is complementary to equality):

$$[\![t_1 \not\equiv t_2]\!] = \{\mathfrak{f} \in \mathcal{R} \mid \bar{\mathfrak{f}}\,(t_1) \neq \bar{\mathfrak{f}}\,(t_2)\},$$

This definition for a new type of goals fits nicely into the general inductive definition of denotational semantics of an arbitrary goal and preserves its properties, such as completeness condition.

In the operational case we deviate from describing one specific search implementation since there are several distinct ways to embed disequality constraints in the language and we would like to be able to give semantics (and subsequently prove correctness) for all of them. Therefore we base the extended operational semantics on a number of abstract definitions concerning constraint stores for which different concrete implementations may be substituted.

We assume that we are given a set of constraint store objects, which we denote by Ω_σ (indexing every constraint store with some substitution σ and assuming the store and the substitution are consistent with each other), and three following operations:

(1) Initial constraint store Ω_ϵ^{init} (where ϵ is empty substitution), which does not contain any constraints yet.
(2) Adding a disequality constraint to a store $\textbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2)$, which may result in a new constraint store Ω'_σ or a failure $\bot$, if the new constraint store is inconsistent with the substitution σ.
(3) Updating a substitution in a constraint store $\textbf{update}\,(\Omega_\sigma, \delta)$ to intergate a new substitution δ into the current one, which may result in a new constraint store $\Omega'_{\sigma\delta}$ or a failure $\bot$, if the constraint store is inconsistent with the new substitution.

The change in operational semantics for the language with disequality constraints is now straightforward: we add a constraint store to a basic (leaf) state $\langle g, \sigma, \Omega_\sigma, n\rangle$, as well as in the label form $(\sigma, \Omega_\sigma, n)$, and this store is simply threaded through all the rules, except those for equality. We change the rules for equality using **update** operation and add the rules for disequality constraint using **add**. In both cases, the search in the current branch is pruned if these primitives return $\bot$.

$$\langle t_1 \equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{\circ} \Diamond,\ \nexists\ mgu\,(t_1, t_2, \sigma) \qquad [\textsc{UnifyFailMGU}]$$

$$\langle t_1 \equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{\circ} \Diamond,\ mgu\,(t_1, t_2, \sigma) = \delta,\ \textbf{update}\,(\Omega_\sigma, \delta) = \bot \qquad [\textsc{UnifyFailUpdate}]$$

$$\langle t_1 \equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{(\sigma\delta,\, \Omega'_{\sigma\delta},\, n)} \Diamond,\ mgu\,(t_1, t_2, \sigma) = \delta,\ \textbf{update}\,(\Omega_\sigma, \delta) = \Omega'_{\sigma\delta} \qquad [\textsc{UnifySuccess}]$$

$$\langle t_1 \not\equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{\circ} \Diamond,\ \textbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \bot \qquad [\textsc{DiseqFail}]$$

$$\langle t_1 \not\equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{(\sigma,\, \Omega'_\sigma,\, n)} \Diamond,\ \textbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \Omega'_\sigma \qquad [\textsc{DiseqSucess}]$$

The initial state naturally contains an initial constraint store $\left\langle g, \varepsilon, \Omega^{init}_\epsilon, n\right\rangle$.

To state the soundness and completeness result now we need to revise our definition of the denotational analog of an answer $(\sigma, \Omega_\sigma, n)$ since we have to take into account the restrictions which a constraint store Ω_σ encodes. To do this we need one more abstract definition — a denotational interpretation of a constraint store $[\![\Omega_\sigma]\!]$ as a set of representing functions. We prove the soundness and completeness w.r.t. this interpretation and expect it to adequately reflect how the restrictions of constraint stores in the answers are presented. The denotational analog of operational semantics for an arbitrary extended state is then redefined as follows.

$$[\![s]\!]_{op} = \cup_{(\sigma, \Omega_\sigma, n) \in \mathcal{T}r_s} [\![\sigma]\!] \cap [\![\Omega_\sigma]\!]$$

The statement of the soundness and completeness theorem stays the same with regard to this updated definitions of semantics and denotational analog.

Theorem 2 (Operational semantics soundness and completeness for extended language). *For any specification $\{\dots\}$ g, for which the indices of all free variables in g are limited by some number n*

$$[\![\langle g, \epsilon, \Omega^{init}_\epsilon, n\rangle]\!]_{op} =_n [\![\{\dots\}\ g]\!].$$

To be able to prove it we, of course, need certain requirements for the given operations on constraint stores. We came up with the following list of sufficient conditions for soundness and completeness.

(1) $[\![\Omega^{init}_\epsilon]\!] = \{\mathfrak{f} : \mathcal{A} \mapsto \mathcal{D}\}$;
(2) $\textbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \Omega'_\sigma \implies [\![\Omega_\sigma]\!] \cap [\![t_1 \not\equiv t_2]\!] \cap [\![\sigma]\!] = [\![\Omega'_\sigma]\!] \cap [\![\sigma]\!]$;
(3) $\textbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \bot \implies [\![\Omega_\sigma]\!] \cap [\![t_1 \not\equiv t_2]\!] \cap [\![\sigma]\!] = \varnothing$;
(4) $\textbf{update}\,(\Omega_\sigma, \delta) = \Omega'_{\sigma\delta} \implies [\![\Omega_\sigma]\!] \cap [\![\sigma\delta]\!] = [\![\Omega'_{\sigma\delta}]\!] \cap [\![\sigma\delta]\!]$;
(5) $\textbf{update}\,(\Omega_\sigma, \delta) = \bot \implies [\![\Omega_\sigma]\!] \cap [\![\sigma\delta]\!] = \varnothing$.

These conditions state that given denotational interpretation and given operations on constraint stores are adequate to each other.

Condition 1 states that interpretation of the initial constraint store is the whole domain of representing function since it does not impose any restrictions.

Condition 2 states that when we add a constraint to a store Ω_σ the interpretation of the result contains exactly those functions which simultaneously belong to the interpretation of the store Ω_σ and satisfy the constraint if we consider only extensions of the substitution σ.

Condition 3 states that addition could fail only if no such functions exist.

Conditions 4 state that the result of updating a store with an additional substitution should have the same interpretation if we consider only extensions of the updated substitution.

Condition 5 states that update could fail only if no such functions exist.

The conditions 2-5 describe exactly what we need to prove the soundness and completeness for base goals (equality and disequality); at the same time, since these conditions have relatively simple intuitive meaning in terms of these two operations they are expected to hold naturally in all reasonable implementations of constraint stores.

We can prove that this is enough for soundness and completeness to hold for an arbitrary goal. However, contrary to our expectations, the existing proof can not be just reused for all non-basic types of goals and has to be modified significantly in the case of **fresh**. Specifically, we need one additional condition on constraint store in state $(\sigma, n, \Omega_\sigma)$: only the values on the first n fresh variables determine whether a representing function belongs to the denotational semantics $[\![\sigma]\!] \cap [\![\Omega_\sigma]\!]$ of the state (note the similarity to the completeness condition). Luckily, we can infer this property for all states that can be constructed by our operational semantics from the sufficient conditions above.

Thus for an arbitrary implementation, we need to give a formal definition of constraint store object and its denotational interpretation, provide three operations for it and prove five conditions on them, and by this, we ensure that for arbitrary specification the interpretations of all solutions found by the search in this version of MiniKanren will cover exactly the mathematical model of this specification.

As well as our previous development this extension is certified in Coq[1]. We describe operational semantics and its soundness and completeness as modules parametrized by the definitions of constraint stores and proofs of the sufficient conditions for them.

4 CONCRETE IMPLEMENTATIONS

In this section, we define two concrete implementations of constraint stores which can be incorporated in operational semantics: the trivial one and the one, which is close to existing real implementation in a certain version of miniKanren [Alvis et al. 2011]. We prove that they satisfy the sufficient conditions for search completeness from the previous section. Both implementations are certified in Coq, which allowed us to extract two correct-by-construction interpreters for miniKanren with disequality constraints.

4.1 Trivial Implementation

This trivial implementation simply stores all pairs of terms, which the search encounters, in a multiset and never uses them:

$$\Omega_\sigma \subset_m \mathcal{T} \times \mathcal{T}$$

$$\Omega_\epsilon^{init} = \varnothing$$

$$\textbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \Omega_\sigma \cup \{(t_1, t_2)\}$$

$$\textbf{update}\,(\Omega_\sigma, \delta) = \Omega_\sigma$$

The interpretation of such constraint store is the set of all representing functions that does not equate terms in any pair:

$$[\![\Omega_\sigma]\!] = \{\mathfrak{f} \colon \mathcal{A} \mapsto \mathcal{D} \mid \forall (t_1, t_2) \in \Omega_\sigma,\ \bar{\mathfrak{f}}\,(t_1) \neq \bar{\mathfrak{f}}\,(t_2)\}$$

This is a correct implementation (although for the full implementation we should find a way to present restrictions stored this way in answers adequately) and it satisfies the sufficient conditions for completeness

[1]https://github.com/dboulytchev/miniKanren-coq/tree/disequality

trivially, but it is not very practical. In particular, it does not use information acquired from disequalities to halt the search in case of contradiction and it can return contradictory answers with the final disequality constraint violated by the final substitution (such as $([\alpha_0 \mapsto 5], [\alpha_0 \neq 5], 1)$): since such answers have empty interpretations, their presence does not affect search completeness.

4.2 Realistic Implementation

This implementation is more similar to those in existing miniKanren implementations and takes an approach that is close to one described is [Alvis et al. 2011].

In this version, every constraint is represented as a substitution containing variable bindings which should *not* be satisfied.

$$\Omega_\sigma \subset_m \Sigma$$

So if a constraint store Ω_σ contains a substitution ω the set of representing functions prohibited by it is $[\![\sigma\omega]\!]$, which provides the following denotational interpretation for a constraint store:

$$[\![\Omega_\sigma]\!] = \bigcap_{\omega \in \Omega_\sigma} \overline{[\![\sigma\omega]\!]}$$

We start with an empty store

$$\Omega_\epsilon^{init} = \varnothing$$

When we encounter a disequality for two terms we try to unify them and update constraint store depending on the result of unification:

$$\mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \begin{cases} \Omega_\sigma & \nexists mgu(t_1\sigma, t_2\sigma) \\ \bot & mgu(t_1\sigma, t_2\sigma) = \epsilon \\ \Omega_\sigma \cup \{\omega\} & mgu(t_1\sigma, t_2\sigma) = \omega \neq \epsilon \end{cases}$$

If the terms are not unifiable, there is no need to change the constraint store. If they are unified by current substitution the constraint is already violated and we signal a failure. Otherwise, the most general unifier is an appropriate representation of this constraint.

When updating a constraint store with an additional substitution δ we try to update each individual constraint substitution by treating it as a list of pairs of terms that should not be unified (the first element of each pair is a variable), applying δ to these terms and trying to unify all pairs simultaniously:

$$\mathbf{update_{constr}}\,([x_1 \mapsto t_1, \dots, x_k \mapsto t_k], \delta) = mgu([\delta(x_1), \dots, \delta(x_k)], [t_1\delta, \dots, t_k\delta])$$

We construct the updated constraint store from the results of all constraint updates:

$$\mathbf{update}\,(\Omega_\sigma, \delta) = \begin{cases} \bot & \exists \omega \in \Omega_\sigma : \mathbf{update_{constr}}\,(\omega, \delta) = \epsilon \\ \{\omega' \mid \mathbf{update_{constr}}\,(\omega, \delta) = \omega' \neq \bot,\ \omega \in \Omega_\sigma\} & otherwise \end{cases}$$

If any constraint is violated by the additional substitution we signal a failure, otherwise we take in the store the updated constraints (and some constraints are thrown away as they can no longer be violated).

We proved the sufficient conditions for completeness for this implementation, too, but it required us to prove first that all substitutions constructed by miniKanren search have a specific form. Namely, a current subsitution σ at any point of the search (started from the initial state) is always *narrowing* — which means that $\mathcal{VR}an\,(\sigma) \cap \mathcal{D}om\,(\sigma) = \varnothing$ — and every time a current substitution σ is updated by composing with some substitution δ (in rule [UnifySuccess]) this substitution is *extending* — which means that $\mathcal{D}om\,(\delta) \cap \mathcal{D}om\,(\sigma) = \varnothing \wedge \mathcal{VR}an\,(\delta) \cap \mathcal{D}om\,(\sigma) = \varnothing$.

5 APPLICATIONS

In addition to verification of correctness of different implementations of disequality constraints we can use our framework to formally state and prove some of its other important properties. Thanks to our completeness result, we can do it in the denotational context, where the reasoning is much easier.

For example, we can specify contradictory answers with empty interpretation, which we pointed out for the trivial implementation from the previous section, and prove that there are no such answers in the realistic implementation if and only if there are infinitely many constructors in the language. So, for the realistic implementation the following holds iff the set of constructors is infinite:

Lemma 1. *For any goal g, if all free variables in it belong to the set $\{\alpha_1, \dots, \alpha_n\}$, then*

$$\forall (\sigma, \Omega_\sigma, n_r) \in Tr_{\langle g, \epsilon, \Omega_\epsilon^{init}, n \rangle}, \quad [\![\sigma]\!] \cap [\![\Omega_\sigma]\!] \neq \varnothing.$$

The proof is based on the following lemma about combining constraints, which we can prove we can prove when there are infinitely many constructors (and otherwise it is not true).

Lemma 2. *If for a finite constraint store Ω_σ*

$$\forall \omega \in \Omega_\sigma, [\![\sigma]\!] \cap [\![\omega]\!] \neq \varnothing,$$

then

$$[\![\sigma]\!] \cap [\![\Omega_\sigma]\!] \neq \varnothing.$$

Another example of application is the justification of optimizations in constraint store implementation. For example, the following obvious (in denotational context) statement allows deleting subsumed constraints in the realistic implementation.

Lemma 3. *For any constraint store Ω_σ and two constraint substitutions ω and ω', if*

$$\exists \tau, \omega' = \omega\tau$$

then

$$[\![\Omega_\sigma \cup \{\omega, \omega'\}]\!] = [\![\Omega_\sigma \cup \{\omega\}]\!].$$

6 CONCLUSION

In this paper we presented an extended version of formal semantics for miniKanren which supports disequality constraints. The semantics is parametrized by an exact implementation of constraint stores and allows us to ensure the correctness of different implementations in a unified way, using the given set of sufficient conditions.

REFERENCES

Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. 2011. cKanren: miniKanren with Constraints. In *Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming*.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer. https://doi.org/10.1007/978-3-662-07964-5

Hubert Comon-Lundh. 1991. Disunification: A Survey. In *Computational Logic - Essays in Honor of Alan Robinson.*

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

Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*.

Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*. 96–107. https://doi.org/10.1145/2989225.2989230

Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. 1998. The semantics of constraint logic programs. *The Journal of Logic Programming* 37, 1 (1998), 1 – 46. https://doi.org/10.1016/S0743-1066(98)10002-X

Robert M. Keller. 1976. Formal Verification of Parallel Programs. *Commun. ACM* 19, 7 (1976), 371–384. https://doi.org/10.1145/360248.360251

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). (2005), 192–203. https://doi.org/10.1145/1086365.1086390

Ramana Kumar. 2010. Mechanising Aspects of miniKanren in HOL. Bachelor Thesis, The Australian National University.

John W. Lloyd. 1984. *Foundations of Logic Programming, 1st Edition*. Springer.

Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2017. Typed Relational Conversion. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*. 39–58. https://doi.org/10.1007/978-3-319-89719-6_3

Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*. 18:1–18:13. https://doi.org/10.1145/3236950.3236958

Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2019. Certified Semantics for miniKanren. In *miniKanren and Relational Programming Workshop*.