Certified Semantics for miniKanren*

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

We present two formal semantics for the core miniKanren. First, we give denotational variant which corresponds to the minimal Herbrand model for definite logic programs. Second, we present operational semantics which models interleaving, and prove its soundness and completeness w.r.t. denotational semantics. Our development is supported by formal Coq specification, thus making it certified.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; **Denotational semantics**; **Operational semantics**;

Additional Key Words and Phrases: Relational programming, denotational semantics, operational semantics, certified programming

1 INTRODUCTION

The introductory book on miniKanren [Friedman et al. 2005] describes the language by means of an evolving set of examples. In the series of follow-up papers [Alvis et al. 2011; Byrd and Friedman 2007; Hemann and Friedman 2013, 2015; Hemann et al. 2016; Swords and Friedman 2013] various extensions of the language were presented with their semantics explained in terms of Scheme implementation. We argue that this style of semantic definition is fragile and not self-evident since it requires the knowledge of semantics of concrete implementation language. In addition the justification of important properties of relational programs (for example, refutational completeness [Byrd 2009]) becomes cumbersome. In the area of programming languages research a formal definition for the semantics of language of interest is a *de-facto* standard, and in our opinion in its current state miniKanren deviates from this standard.

There were some previous attempts to define a formal semantics for miniKanren. Lozov et al. [2017] present a variant of nondeterministic operational semantics, and Rozplokhas and Boulytchev [2018] use another variant of finite-set semantics. None of them was capable of reflecting the distinctive property of miniKanren search — *interleaving* [Kiselyov et al. 2005], thus deviating from the conventional understanding of the language.

In this paper we present a formal semantics for core miniKanren and prove some its basic properties. First, we define denotational semantics similar to the least Herbrand model for definite logic programs [Lloyd 1984]; then we describe operational semantics with interleaving in terms of labeled transition system. Finally, we prove the soundness and completeness of the operational semantics w.r.t the denotational one. We support our development

*This work was partially suppored by the grant 18-01-00380 from The Russian Foundation for Basic Research

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

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

Fig. 1. The syntax of the source language

$$
\begin{array}{c}
\mathcal{FV}\,(x) = \{x\} \\
\mathcal{FV}\,(C_i^{k_i}\,(t_1, \dots, t_k)) = \bigcup \mathcal{FV}\,(t_i) \\
\\
\mathcal{FV}\,(t_1 \equiv t_2) = \mathcal{FV}\,(t_1) \cup \mathcal{FV}\,(t_2) \\
\mathcal{FV}\,(g_1 \wedge g_2) = \mathcal{FV}\,(g_1) \cup \mathcal{FV}\,(g_2) \\
\mathcal{FV}\,(g_1 \vee g_2) = \mathcal{FV}\,(g_1) \cup \mathcal{FV}\,(g_2) \\
\mathcal{FV}\,(\mathbf{fresh}\ x\ .\ g) = \mathcal{FV}\,(g) \setminus \{x\} \\
\mathcal{FV}\,(R_i^{k_i}\,(t_1, \dots, t_k)) = \bigcup \mathcal{FV}\,(t_i)
\end{array}
$$

Fig. 2. Free variables in terms and goals

with a formal specification using Coq [Bertot and Castéran 2004] proof assistant[1], thus outsourcing the burden of proof checking to the automatic tool.

The paper organized as follows. In Section 2 we give the syntax of the language, describe its semantics informally and discuss some examples. Section 3 contains the description of denotational semantics for the language, and Section 4 — the operational semantics. In Section 5 we overview the certified proof for soundness and completeness of operational semantics. The final section concludes.

2 THE LANGUAGE

In this section we introduce the syntax of the language we use throughout the paper, describe the informal semantics and give some examples.

The syntax of the language is shown on Figure 1. First, we fix a set of constructors $\mathcal{C}$ 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

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

five types of goals: *unification* of terms, conjunction and disjunction of goals, fresh variable introduction and invocation of some relational definition. Thus, unification 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 "**fresh** $x\,y$ . . . . g" instead of "**fresh** x . **fresh** y . . . . 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 definition for set of free variables for both terms and goals is given on Figure 2; as "**fresh**" is the sole binding construct the definition is rather trivial. The language we defined is first-order, as goals can not be passed as parameters, returned or constructed at runtime.

We now informally describe how relational search works. As we said, a goal represents a search procedure. This procedure takes a *state* as input and returns a stream of states; a state (among other information) contains a substitution which maps semantic variables into terms over semantic variables. Then five types of scenarios are possible (dependent on the type of the goal):

- Unification "$t_1 \equiv t_2$" unifies terms t_1 and t_2 in the context of the substitution in the current state. If terms are unifiable, then their MGU is integrated into the substitution, and one-element stream is returned; otherwise the result is an empty stream.
- Conjunction "$g_1 \wedge g_2$" applies g_1 to the current state and then applies g_2 to the each element of the result, concatenating the streams.
- Disjunction "$g_1 \vee g_2$" applies both its goals to the current state independently and then concatenates the results.
- Fresh construct "**fresh** x . g" allocates a new semantic variable α, substitutes all free occurrences of x in g with α, and runs the goal.
- Invocation "$R_i^{k_i}$ $(t_1,...,t_{k_i})$" finds a definition for relational symbol $R_i^{k_i} = \lambda x_1 \dots x_{k_i} \,.\, g_i$, substitutes all free occurrences of formal parameter x_j in g_i with term t_j (for all j) and runs the goal in the current state.

We stipulate, that the top-level goal is preceded by an implicit "**fresh**" construct, which binds all its free variables, and the final substitutions for these variables constitute the result of the goal evaluation.

Conjunction and disjunction form a monadic [Wadler 1995] interface with conjunction playing role of "bind" and disjunction — of "mplus". In this description we swept a lot of important details under the carpet — for example, in actual implementations the components of disjunction are not evaluated in isolation, but both disjuncts are being evaluated incrementally with the control passing from one disjunct to another (*interleaving*); instead streams the implementation can be based on "ferns" [Byrd et al. [n. d.]] to defer divergent computations, etc.

As an example consider the following specification:

```
appendᵒ = λ x y xy .
  ((x ≡ Nil) ∧ (xy ≡ y)) ∨
  (fresh h t ty .
     (x  ≡ Cons (h, t))  ∧
     (xy ≡ Cons (h, ty)) ∧
     (appendᵒ y t ty)
  );
reversᵒ = λ x y .
  ((x ≡ Nil) ∧ (y ≡ Nil)) ∨
  (fresh h t t' .
     (x ≡ Cons (h, t)) ∧
```

```
      (append$^o$ t' (Cons (h, Nil)) y) ∧
      (revers$^o$ t t')
   );
revers$^o$ x x
```

Here we defined two relational symbols — "append^o" and "revers^o", — and specified a top-level goal "revers^o x x". The symbol "append^o" defines a relational concatenation of lists — it takes three arguments and performs a case analysis on the first one. If the first one is an empty list ("Nil"), then the second and the third arguments are unified. Otherwise the first argument is deconstructed into a head "h" and a tail "t", and the tail is concatenated with the second argument using a recursive call to "append^o" and additional variable "ty", which represents the concatenation of "t" and "y". Finally, we unify "Cons (h, ty)" with "xy" to form a final constraint. Similarly, "revers^o" defines relational list reversing. The top-level goal represents a search procedure for all lists "x", which are stable under reversing, i.e. represent palindromes. Running it results in an infinite stream of substitutions:

$\alpha \mapsto$ Nil
$\alpha \mapsto$ Cons (β_0, Nil)
$\alpha \mapsto$ Cons (β_0, Cons (β_0, Nil))
$\alpha \mapsto$ Cons (β_0, Cons (β_1, Cons (β_0, Nil)))
...

where "α" — a *semantic* variable, corresponding to "x", "β_i" — free semantics variables.

The notions above can be formalized in Coq in a natural way using inductive data types. We have made a few non-essential simplifications and modifications for the sake of convenience.

Specifically, we restrict the arities of constructors to be either zero or two:

```
Inductive term : Set :=
| Var : var → term
| Cst : con_name → term
| Con : con_name → term → term → term.
```

Here "var" and "con_name" — types representing variables and constructor names, whose definitions we omitted for the sake of brevity. Similarly, we restrict relations to always have exactly one argument:

```
Definition rel : Set := term → goal.
```

These restrictions do not make the language less expressive in any way since we can represent a sequence of terms as a list using constructors Nil^0 and Cons^2.

We also introduce one additional type of goals — *failure* — for deliberately unsuccessful computation (empty stream). As a result, the definition of goals looks as follows:

```
Inductive goal : Set :=
| Fail   : goal
| Unify  : term → term → goal
| Disj   : goal → goal → goal
| Conj   : goal → goal → goal
| Fresh  : (var → goal) → goal
| Invoke : rel_name → term → goal.
```

$$
\begin{array}{rcll}
x\,[t/x] & = & t & \\
y\,[t/x] & = & y & y \neq x \\
C_i^{k_i}\,(t_1, \dots, t_{k_i})\,[t/x] & = & C_i^{k_i}\,(t_1\,[t/x], \dots, t_{k_i}\,[t/x]) & \\
(t_1 \equiv t_2)\,[t/x] & = & t_1\,[t/x] \equiv t_2\,[t/x] & \\
(g_1 \wedge g_2)\,[t/x] & = & g_1\,[t/x] \wedge g_2\,[t/x] & \\
(g_1 \vee g_2)\,[t/x] & = & g_1\,[t/x] \vee g_2\,[t/x] & \\
(\mathbf{fresh}\ x\,.\,g)\,[t/x] & = & \mathbf{fresh}\ x\,.\,g & \\
(\mathbf{fresh}\ y\,.\,g)\,[t/x] & = & \mathbf{fresh}\ y\,.\,(g\,[t/x]) & y \neq x \\
(R_i^{k_i}\,(t_1, \dots, t_{k_i})\,[t/x] & = & R_i^{k_i}\,(t_1\,[t/x], \dots, t_{k_i}\,[t/x]) &
\end{array}
$$

Fig. 3. Substitutions for terms and goals

Note that in our formalization we use the higher-order abstract syntax for variable binding [Pfenning and Elliott 1988]. We preferred it to the first-order syntax because it gives us the ability to use substitution and inductive principle provided by Coq. However, we still need to carefully ensure some expected properties on the structure of syntax trees. For example, we should require that the definitions of relations do not contain unbound variables:

```
Definition closed_goal_in_context (c : list var) (g : goal) : Prop :=
  ∀ n, is_fv_of_goal n g → In n c.

Definition closed_rel (r : rel) : Prop :=
  ∀ (arg : term), closed_goal_in_context (fv_term arg) (r arg).

Definition def : Set := {r : rel | closed_rel r}.
```

In the snippet above "def" corresponds to a set of relational symbol definitions in a specification.

We set an arbitrary environment (a map from relational symbol to the definition of relation) to use further throughout the formalization. Failure goals allow us to define it as a total function:

```
Definition env : Set := rel_name → def.

Variable Prog : env.
```

3 DENOTATIONAL SEMANTICS

In this section we present a denotational semantics for the language we defined above. We use a simple set-theoretic approach which can be considered as an analogy to the least Herbrand model for definite logic programs [Lloyd 1984]. Strictly speaking, instead of developing it from scratch we could have just described the conversion of specifications into definite logic form and took their least Herbrand model. However, in that case we would still need to define the least Herbrand model semantics for definite logic programs in a certified way. In addition, while for this concrete language the conversion to definite logic form is trivial, it may become less trivial for its extensions (with, for examples, nominal constructs [Byrd and Friedman 2007]) which we plan to do in future.

To motivate further development, we first consider the following example. Let us have the following goal:

```
x ≡ Cons (y, z)
```

There are three free variables, and solving the goal delivers us the following single answer:

$\alpha \mapsto$ `Cons` $(\beta,\ \gamma)$

where semantic variables α, β and γ correspond to the syntactic ones "x", "y", "z". The goal does not put any constraints on "y" and "z", so there are no bindings for "β" and "γ" in the answer. This answer can be seen as the following ternary relation over the set of all ground terms:

$$\{(\texttt{Cons}\ (\beta, \gamma), \beta, \gamma) \mid \beta \in \mathcal{D},\ \gamma \in \mathcal{D}\} \subset \mathcal{D}^3$$

The order of "dimensions" is important, since each dimension corresponds to a certain free variable. Our main idea is to 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*. Thus, we may reformulate the same relation as

$$\{(\mathfrak{f}\,(\alpha), \mathfrak{f}\,(\beta), \mathfrak{f}\,(\gamma)) \mid \mathfrak{f} \in [\![\alpha \equiv \texttt{Cons}\ (\beta, \gamma)]\!]\}$$

where we use conventional semantic brackets "$[\![\bullet]\!]$" to denote the semantics. For the top-level goal we need to substitute its free syntactic variables with distinct semantic ones, calculate the semantics, and build the explicit representation for the relation as shown above. The relation, obviously, does not depend on concrete choice of semantic variables, but depends on the order in which the values of representing functions are tupled. This order can be conventionalized, which gives us a completely deterministic semantics.

Now we implement this idea. First, for a representing function

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

we introduce its homomorphic extension

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

which maps terms to terms:

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

Let us have two terms $t_1, t_2 \in \mathcal{T}_{\mathcal{A}}$. If there is a unifier for t_1 and t_2 then, clearly, there is a substitution θ which turns both t_1 and t_2 into the same *ground* term (we do not require θ to be the most general). Thus, θ maps (some) ground variables into ground terms, and its application to $t_{1(2)}$ is exactly $\overline{\theta}(t_{1(2)})$. This reasoning can be performed in the opposite direction: a unification $t_1 \equiv t_2$ defines the set of all representing functions $\mathfrak{f}$ for which $\overline{\mathfrak{f}}(t_1) = \overline{\mathfrak{f}}(t_2)$.

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 which 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.

We formulate the following important *completeness condition* for the semantics of a goal g:

$$\forall \alpha \notin FV(g)\ \forall d \in \mathcal{D}\ \forall \mathfrak{f} \in [\![g]\!]\ \exists \mathfrak{f}' \in [\![g]\!]\ :\ \mathfrak{f}'(\alpha)\ = d \land \forall \beta \neq \alpha :\ \mathfrak{f}'(\beta)\ = \mathfrak{f}(\beta)$$

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. It can be proven that the semantics of goals always satisfy this condition.

We remind conventional notions of pointwise modification of a function

$$f[x \leftarrow v](z) = \begin{cases} f(z) & , \quad z \neq x \\ v & , \quad z = x \end{cases}$$

and substitution of a free variable with a term in terms and goals (see Figure 3).

For a representing function $\mathfrak{f} : \mathcal{A} \to \mathcal{D}$ and a semantic variable α we define the following *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 $\mathfrak{F} \subseteq \mathcal{A} \to \mathcal{D}$:

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

Now we are ready to specify the semantics for goals (see Figure 4). We've already given the motivation for the semantics of unification: the condition $\overline{\mathfrak{f}}(t_1) = \overline{\mathfrak{f}}(t_2)$ gives us the set of all (otherwise unrestricted) representing functions which "equate" terms t_1 and t_2. Set union and intersection provide a conventional interpretation for disjunction and conjunction of goals, and the semantics of relational invocation reduces to the application of corresponding function from the environment. The only interesting case is "**fresh** x . g". First, we take an arbitrary semantic variable α, not free in g, and substitute x with α. Then we calculate the semantics of $g[\alpha/x]$. The interesting part is the next step: as x can not be free in "**fresh** x . g", we need to generalize the result over α since in our model the semantics of a goal specifies a relation over its free variables. We introduce some nondeterminism, by choosing arbitrary α, but it can be proven by structural induction, that with different choices of free variable, semantics of a goal won't change. Consider the following example:

fresh `y . (`α` ≡ y) ∧ (y ≡ Zero)`

As there is no invocations involved, we can safely omit the environment. Then:

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

Fig. 4. Denotational semantics of goals

$$
\begin{array}{lcr}
[\![\mathbf{fresh}\ \ \mathtt{y}\ \ .\ \ (\alpha \equiv \mathtt{y}) \wedge (\mathtt{y}\ \equiv \mathtt{Zero})]\!] & = & (\text{by } \textsc{Fresh}_D) \\
([\![(\alpha \equiv \beta) \wedge (\beta \equiv \mathtt{Zero})]\!]) \uparrow \beta & = & (\text{by } \textsc{Conj}_D) \\
([\![\alpha \equiv \beta]\!]\ \cap\ [\![\beta \equiv \mathtt{Zero})]\!]) \uparrow \beta & = & (\text{by } \textsc{Unify}_D) \\
(\{\mathfrak{f} \mid \bar{\mathfrak{f}}\,(\alpha) = \bar{\mathfrak{f}}\,(\beta)\}\ \cap\ \{\mathfrak{f} \mid \bar{\mathfrak{f}}\,(\beta) = \bar{\mathfrak{f}}\,(\mathtt{Zero})\}) \uparrow \beta & = & (\text{by the definition of ``}\bar{\mathfrak{f}}\text{''}) \\
(\{\mathfrak{f} \mid \mathfrak{f}\,(\alpha) = \mathfrak{f}\,(\beta)\}\ \cap\ \{\mathfrak{f} \mid \mathfrak{f}\,(\beta) = \mathtt{Zero}\}) \uparrow \beta & = & (\text{by the definition of ``}\cap\text{''}) \\
(\{\mathfrak{f} \mid \mathfrak{f}\,(\alpha) = \mathfrak{f}\,(\beta) = \mathtt{Zero}\}) \uparrow \beta & = & (\text{by the definition of ``}\uparrow\text{''}) \\
\{\mathfrak{f} \mid \mathfrak{f}\,(\alpha) = \mathtt{Zero}, \mathfrak{f}\,(\beta) = d, d \in \mathcal{D}\} & = & (\text{by the totality of representing functions}) \\
\{\mathfrak{f} \mid \mathfrak{f}\,(\alpha) = \mathtt{Zero}\} & &
\end{array}
$$

In the end we've got the set of representing functions, each of which restricts only the value of free variable α.

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 have to 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. We consider the following function

$$\mathcal{F} : (\mathcal{T_A}^* \to 2^{\mathcal{A} \to \mathcal{D}})^n \to (\mathcal{T_A}^* \to 2^{\mathcal{A} \to \mathcal{D}})^n$$

which represents a semantic for the set of definitions abstracted over themselves. The definition of this function is rather standard:

$$
\begin{array}{rcl}
\mathcal{F}\,(p_1, \dots, p_n) & = & (t_1^1 \dots t_{k_1}^1 \mapsto [\![g^1\,[t_1^1/x_1^1, \dots, t_{k_1}^1/x_{k_1}^1]]\!]_\Gamma, \\
& & \quad \dots \\
& & \quad t_1^n \dots t_{k_n}^n \mapsto [\![g^n\,[t_1^n/x_1^n, \dots, t_{k_n}^n/x_{k_n}^n]]\!]_\Gamma) \\
& & \quad \text{where } \Gamma\, R_i = p_i
\end{array}
$$

Here p_i is a semantic function for i-th definition; we build an environment Γ which associates each relational symbol R_i with p_i and construct a n-dimensional vector-function, where i-th component corresponds to a function which calculates the semantics of i-th relational definition application to terms w.r.t. the environment Γ. Finally, we take the least fixed point of $\mathcal{F}$ and define the top-level environment as follows:

$$\Gamma_0\, R_i = (fix\ \mathcal{F})\,[i]$$

where "$[i]$" denotes the i-th component of a vector-function.

The least fixed point exists by Knaster-Tarski [Tarski 1955] theorem — the set $(\mathcal{T}_{\mathcal{A}}{}^{*} \to 2^{\mathcal{A} \to \mathcal{D}})^n$ forms a complete lattice, and $\mathcal{F}$ is monotonic.

To formalize denotational semantics in Coq we can define representing functions simply as Coq functions:

```
Definition repr_fun : Set := var → ground_term.
```

We define the semantics via inductive proposition "`in_denotational_sem_goal`" such that

$$\forall g, \mathfrak{f} \ : \ \texttt{in\_denotational\_sem\_goal}\ g\ \mathfrak{f} \iff \mathfrak{f} \in [\![g]\!]_{\Gamma}$$

The definition is as follows:

```
Inductive in_denotational_sem_goal : goal → repr_fun → Prop :=
| dsgUnify  : ∀ f t1 t2, apply_repr_fun f t1 = apply_repr_fun f t2 →
                         in_denotational_sem_goal (Unify t1 t2) f

| dsgDisjL  : ∀ f g1 g2, in_denotational_sem_goal g1 f →
                         in_denotational_sem_goal (Disj g1 g2) f

| dsgDisjR  : ∀ f g1 g2, in_denotational_sem_goal g2 f →
                         in_denotational_sem_goal (Disj g1 g2) f

| dsgConj   : ∀ f g1 g2, in_denotational_sem_goal g1 f →
                         in_denotational_sem_goal g2 f →
                         in_denotational_sem_goal (Conj g1 g2) f

| dsgFresh  : ∀ f fn a fg, (~ is_fv_of_goal a (Fresh fg)) →
                           in_denotational_sem_goal (fg a) fn →
                           (∀ x, x <> a → fn x = f x) →
                           in_denotational_sem_goal (Fresh fg) f

| dsgInvoke : ∀ r t f, in_denotational_sem_goal (proj1_sig (Prog r) t) f →
                       in_denotational_sem_goal (Invoke r t) f.
```

Here we refer to a fixpoint "`apply_repr_fun`" which calculates the extension "$\overline{\bullet}$" for a representing function, and inductive proposition "`is_fv_of_goal`" which encodes the set of free variables for a goal.

Recall that the environment "`Prog`" maps every relational symbol to the definition of relation, which is a pair of a function from terms to goals and a proof that it has no unbound variables. So in the last case "`(proj1_sig (Prog r) t)`" simply takes the body of the corresponding relation; thus "`Prog`" in Coq specification plays role of a global environment Γ.

It is interesting that in Coq implementation we do not need to refer to Tarski-Knaster theorem explicitly since the least fixpoint semantic is implicitly provided by inductive definitions.

4 OPERATIONAL SEMANTICS

In this section we describe operational semantics of miniKanren, which corresponds to the known implementations with interleaving search. The semantics will be given in the form of labeled transition system (LTS). From now on we assume the set of semantic variables to be linearly ordered ($\mathcal{A} = \{\alpha_1, \alpha_2, \dots\}$).

We introduce the notion of substitution

$$\sigma : \mathcal{A} \to \mathcal{T}_{\mathcal{A}}$$

as a (partial) mapping from semantic variables to terms over the set of semantic variables. We denote Σ the set of all substitutions, $\mathcal{D}om\,(\sigma)$ — the domain for a substitution σ, $\mathcal{VR}an\,(\sigma) = \bigcup_{\alpha \in \mathcal{D}om\,(\sigma)} \mathcal{FV}\,(\sigma\,(\alpha))$ — its range (the set of all free variables in the image).

The states in the transition system have the following shape

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

As we will see later, an evaluation of a goal is separated into elementary steps, and these steps are performed interchangeably for different subgoals. Thus, 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 evaluation, and the following well-formedness condition:

Definition 1. *Well-formedness condition for extended states:*

- *$\diamond$ is well-formed;*
- *$\langle g, \sigma, n \rangle$ is well-formed iff $\mathcal{FV}\,(g) \cup \mathcal{D}om\,(\sigma) \cup \mathcal{VR}an\,(\sigma) \subset \{\alpha_1, \dots, \alpha_n\}$;*
- *$s_1 \oplus s_2$ is well-formed iff s_1 and s_2 well-formed;*
- *$s \otimes g$ is well-formed iff s is well-formed and for all leaf triplets $\langle \_, \_, n \rangle$ in s $\mathcal{FV}\,(g) \subseteq \{\alpha_1, \dots, \alpha_n\}$.*

Informally the well-formedness restricts the set of states to those in which all goals use only allocated variables. Finally, we define the set of labels:

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

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 allocated variables; we ignore it in further explanations.

The transition rules are shown on Figure 5. The first two rules specify the semantics of unification. If two terms are not unifiable under the current substitution σ then the evaluation stops with no answer; otherwise it stops with the answer equal to the most general unifier.

The next two rules describe the steps performed when disjunction (conjunction) is encountered on the top level of the current goal. For disjunction it schedules both goals (using "$\oplus$") for evaluating in the same context as the parent state, for conjunction — schedules the left goal and postpones the right one (using "$\otimes$").

The rule for "**fresh**" substitutes bound syntactic variable with a newly allocated semantic one and proceeds with the goal; no answer provided at this step.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. Operational semantics of interleaving search

The rule for relation invocation finds a corresponding definition, substitutes its formal parameters with the actual ones, and proceeds with the body.

The rest of the rules specify the steps performed during the evaluation of two remaining types of the states — conjunction and disjunction. In all cases the left state is evaluated first. If its evaluation stops with a result then the right state (or goal) is scheduled for evaluation, and the label is propagated. If there is no result then the conjunction evaluation stops with no result (ConjStop) as well while the disjunction evaluation proceeds with the right state (DisjStop).

The last four rules describe *interleaving*, which occurs when the evaluation of the left state suspends with some residual state (with or without an answer). In the case of disjunction the answer (if any) is propagated, and the constituents of the disjunction are swapped (DisjStep, DisjStepAns). In case of conjunction, if the evaluation step in the left conjunct did not provide any answer, the evaluation is continued in the same order since there is still no information to proceed with the evaluation of the right conjunct (ConjStep); if there is some answer, then the disjunction of the right conjunct in the context of the answer and the remaining conjunction is scheduled for evaluation (ConjStepAns).

The introduced transition system is completely deterministic. There was, however, some freedom in choosing the order of evaluation for conjunction and disjunction states. For example, instead of evaluating the left substate first we could choose to evaluate the right one, etc. In each concrete case we would end up with a different (but still deterministic) system which would prescribe different semantics to a concrete goal. This choice reflects the inherent non-deterministic nature of search in relational (and, more generally, logical) programming. However, as long as deterministic search procedures are sound and complete, we can consider them "equivalent"[2].

A derivation sequence for a certain state determines a *trace* — a finite or infinite sequence of answers. We may define a set of finite or infinite sequences X^ω over an alphabet X as a set of functions from natural numbers into a lifted set $X_\perp = X \cup \{\perp\}$:

$$X^\omega = \{\omega : \mathbb{N} \to X_\perp \mid \forall n \in \mathbb{N},\ \omega(n) = \perp \Rightarrow \omega(n+1) = \perp\}$$

Informally speaking, we represent a sequence as a function which maps positions (treated as natural numbers) into the elements of the sequence. We use "$\perp$" to specify that there is no element at given position, and we stipulate, that there are no "holes" in this representation: if there is no element at given position then there are no elements at greater positions as well.

For this representation we may define the empty sequence ϵ and operations of prepending a sequence ω with an element a and taking a suffix of a sequence ω from a position n as follows:

$$\epsilon = i \mapsto \perp$$

$$a\omega = i \mapsto \begin{cases} a & , \quad i = 0 \\ \omega(i-1) & , \quad \text{otherwise} \end{cases}$$

$$\omega[n:] = i \mapsto \omega(n+i)$$

For a given state s a trace $\mathcal{T}r_s \in L^\omega$ is a sequence of labels, defined as follows simultaneously with the sequence of states $\{s_i\}$:

$$\begin{array}{lcl} & s_o = s & \\ \mathcal{T}r_s(n) = a & , \quad s_{n+1} = s' & \text{if} \quad s_n \neq \diamond,\ s_n \xrightarrow{a} x' \\ \mathcal{T}r_s(n) = \perp & , \quad s_{n+1} = \diamond & \text{if} \quad s_n = \diamond \end{array}$$

The trace corresponds to the stream of answers in the reference miniKanren implementations.

To formalize the operational part in Coq we first need to define all preliminary notions from unification theory [Baader and Snyder 2001] which our semantics uses.

[2]There still can be differences in observable behavior of concrete goals under different sound and complete search strategies: a goal can be refutationally complete [Byrd 2009] under one strategy and non-complete under another.

In particular, we need to implement the notion of the most general unifier (MGU). As is it well-known [McBride 2003] all standard recursive algorithms for calculating MGU are not decreasing on argument terms, so we can't define it as a simple recursive function in Coq due to the termination check. There is no such obstacle when we define MGU as a proposition:

```
Inductive MGU : term → term → option subst → Set := ...
```

However, we still need to use a well-founded induction to prove the existence of the most general unifier and its defining properties:

```
Lemma MGU_ex : ∀ t1 t2, {r & MGU t1 t2 r}.

Definition unifier (s : subst) (t1 t2 : term) : Prop := apply_subst s t1 = apply_subst s t2.

Lemma MGU_unifies:
  ∀ t1 t2 s, MGU t1 t2 (Some s) → unifier s t1 t2.

Definition more_general (m s : subst) : Prop :=
  ∃ (s' : subst), ∀ (t : term), apply_subst s t = apply_subst s' (apply_subst m t).

Lemma MGU_most_general :
  ∀ (t1 t2 : term) (m : subst),
    MGU t1 t2 (Some m) →
    ∀ (s : subst), unifier s t1 t2 → more_general m s.

Lemma MGU_non_unifiable :
  ∀ (t1 t2 : term),
    MGU t1 t2 None → ∀ s, ~ (unifier s t1 t2).
```

For this well-founded induction we use the number of free variables in argument terms as a well-founded order on pairs of terms:

```
Definition terms := term * term.

Definition fvOrder (t : terms) := length (union (fv_term (fst t)) (fv_term (snd t))).

Definition fvOrderRel (t p : terms) := fvOrder t < fvOrder p.

Lemma fvOrder_wf : well_founded fvOrderRel.
```

After this preliminary work, the described transition relation can be encoded naturally as an inductively defined proposition (here "state'" stands for an extended state):

```
Inductive eval_step : state → label → state' → Set := ...
```

We state the fact that our system is deterministic through existence and uniqueness of a transition for every state:

```
Lemma eval_step_ex : ∀ (st : state), {l : label & {st' : state' & eval_step st l st'}}.
Lemma eval_step_unique :
  ∀ (st : state) (l1 l2 : label) (st'1 st'2 : state'),
    eval_step st l1 st'1 → eval_step st l2 st'2 → l1 = l2 ∧ st'1 = st'2.
```

To work with (possibly) infinite sequences we use the standard approach in Coq — coinductively defined streams:

```
Context {A : Set}.

CoInductive stream : Set :=
| Nil : stream
| Cons : A → stream → stream.
```

Although the definition of the datatype is coinductive some of its properties we are working with make sense only when defined inductively:

```
Inductive in_stream : A → stream → Prop :=
| inHead : ∀ x t, in_stream x (Cons x t)
| inTail : ∀ x h t, in_stream x t → in_stream x (Cons h t).

Inductive finite : stream → Prop :=
| fNil : finite Nil
| fCons : ∀ h t, finite t → finite (Cons h t).
```

Then we define a trace coinductively as a stream of labels in transition steps and prove that there exists a unique trace from any extended state:

```
Definition trace : Set := @stream label.

CoInductive op_sem : state' → trace → Set :=
| osStop : op_sem Stop Nil
| osState : ∀ st l st' t, eval_step st l st' →
                          op_sem st' t →
                          op_sem (State st) (Cons l t).

Lemma op_sem_ex (st' : state') : {t : trace & op_sem st' t}.

Lemma op_sem_unique :
  ∀ st' t1 t2, op_sem st' t1 → op_sem st' t2 → equal_streams t1 t2.
```

Note, for the equality of streams we need to define a new coinductive proposition instead of using the standard syntactic equality in order for coinductive proofs to work [Chlipala 2013].

One thing we can prove using operational semantics is the *interleaving* properties of disjunction. Specifically, we can prove that a trace for a disjunction is a one-by-one interleaving of streams for its disjuncts:

```
CoInductive interleave : stream → stream → stream → Prop :=
| interNil : ∀ s s', equal_streams s s' → interleave Nil s s'
| interCons : ∀ h t s rs, interleave s t rs → interleave (Cons h t) s (Cons h rs).
```

Lemma `sum_op_sem : ∀ st1 st2 t1 t2 t, op_sem (State st1) t1 →`
```
                                   op_sem (State st2) t2 →
                                   op_sem (State (Sum st1 st2)) t →
                                   interleave t1 t2 t.
```

This allows us to prove the expected properties of interleaving in a more general setting of arbitrary streams:

- the elements of the interleaved stream are exactly those of two interleaved streams;
- the interleaved stream is finite iff both interleaving streams are finite.

The corresponding Coq lemmas are as follows:

Lemma `interleave_in : ∀ s1 s2 s, interleave s1 s2 s →`
```
           ∀ x, in_stream x s ↔ in_stream x s1 ∨ in_stream x s2.
```

Lemma `interleave_finite : ∀ s1 s2 s, interleave s1 s2 s →`
```
           (finite s ↔ finite s1 ∧ finite s2).
```

5 SEMANTICS EQUIVALENCE

Now when we defined two different kinds of semantics for miniKanren we can relate them and show that the results given by these two semantics are the same for any specification. This will actually say something important about the search in the language: since operational semantics describes precisely the behavior of the search and denotational semantics ignores the search and describes what we *should* get from mathematical point of view, by proving their equivalence we establish *completeness* of the search which means that the search will get all answers satisfying the described specification and only those.

But first, we need 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. We can notice that the notion of representing function is close to substitution, with only two differences:

- representing function is total;
- terms in the domain of representing function are ground.

Therefore we can easily extend (perhaps ambiguously) any substitution to a representing function by composing it with an arbitrary representing function and that will preserve all variable dependencies in the substitution. So we can define 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:

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

This allows us to state theorems relating two semantics.

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

$$\llbracket \langle g, \epsilon, n\rangle \rrbracket_{op} \subset \llbracket \{\dots\}\ g \rrbracket.$$

$$\begin{array}{lcl}
[\![\diamond]\!]_\Gamma & = & \varnothing \\
[\![\langle g, \sigma, n\rangle]\!]_\Gamma & = & [\![g]\!]_\Gamma \cap [\sigma] \\
[\![s_1 \oplus s_2]\!]_\Gamma & = & [\![s_1]\!]_\Gamma \cup [\![s_2]\!]_\Gamma \\
[\![s \otimes g]\!]_\Gamma & = & [\![s]\!]_\Gamma \cap [\![g]\!]_\Gamma
\end{array}$$

Fig. 6. Denotational semantics of states

It can be proven by nested induction, but first, we need to generalize the statement so that the inductive hypothesis would be strong enough for the inductive step. To do so, we define denotational semantics not only for goals but for arbitrarily extended states. Note that this definition does not need to have any intuitive interpretation, it is introduced only for proof to go smoothly. The definition of the denotational semantics for extended states is on Figure 6. The generalized version of the theorem uses it:

Lemma 1 (Generalized soundness). *For any top-level environment Γ_0 acquired from some specification, for any well-formed (w.r.t. that specification) extended state s*

$$[\![s]\!]_{op} \subset [\![s]\!]_{\Gamma_0}.$$

It can be proven by induction on the number of steps in which a given answer (more accurately, the substitution that contains it) occurs in the trace. The induction step is proven by structural induction on the extended state s.

It would be tempting to formulate the completeness of operational semantics as the inverse inclusion, but it does not hold in such generality. 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 formulate the completeness with representing functions restricted 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.

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

$$\{\mathfrak{f}|_{\{\alpha_1,\dots,\alpha_n\}} \mid \mathfrak{f} \in [\![\{\dots\}\ g]\!]\} \subset \{\mathfrak{f}|_{\{\alpha_1,\dots,\alpha_n\}} \mid \mathfrak{f} \in [\![\langle g, \epsilon, n\rangle]\!]_{op}\}.$$

Similarly to the soundness, this can be proven by nested induction, but the generalization is required. This time it is enough to generalize it from goals to states of the shape $\langle g, \sigma, n\rangle$. We also need to introduce one more auxiliary semantics — bounded denotational semantics:

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

Instead of always unfolding the definition of a relation for invocation goal, it does so only given number of times. So for a given set of relational definitions $\{R_i^{k_i} = \lambda\, x_1^i \dots x_{k_i}^i\, .\, g_i;\}$ the definition of bounded denotational semantics is exactly the same as in usual denotational semantics, except that for the invocation case:

$$[\![R_i^{k_i}\,(t_1, \dots, t_{k_i})]\!]^{l+1} = [\![g_i[t_1/x_1^i, \dots, t_{k_i}/x_{k_i}^i]]\!]^l$$

It is convenient to define bounded semantics for level zero as an empty set:

$$[\![g]\!]^0 = \varnothing$$

Bounded denotational semantics is an approximation of a usual denotational semantics and it is clear that any answer in usual denotational semantics will also be in bounded denotational semantics for some level:

Lemma 2. $[\![g]\!]_{\Gamma_0} \subset \cup_l [\![g]\!]^l$

Formally it can be proven using the definition of the least fixed point from Tarski-Knaster theorem: the set on the right-hand side is a closed set.

Now the generalized version of the completeness theorem is as follows:

Lemma 3 (Generalized completeness). *For any set of relational definitions, for any level l, for any well-formed (w.r.t. that set of definitions) state $\langle g, \sigma, n\rangle$,*

$$\{\mathfrak{f}|_{\{\alpha_1,\dots,\alpha_n\}} \mid \mathfrak{f} \in [\![g]\!]^l \cap [\sigma]\} \subset \{\mathfrak{f}|_{\{\alpha_1,\dots,\alpha_n\}} \mid \mathfrak{f} \in [\![\langle g, \sigma, n\rangle]\!]_{op}\}.$$

It is proven by induction on the level l. The induction step is proven by structural induction on the goal g.

The proofs of both theorems are certified in Coq, although the proofs for a number of (obvious) technical facts about representing functions and computation of the most general unifier as well as some properties of denotational semantics, proven informally in Section 3, are admitted for now. For completeness we can not just use the induction on proposition `in_denotational_sem_goal`, as it would be natural to expect, because the inductive principle it provides is not flexible enough. So we need to define bounded denotational semantics in our formalization too and perform induction on the level explicitly:

```
Inductive in_denotational_sem_lev_goal : nat → goal → repr_fun → Prop :=
...
| dslgInvoke : ∀ l r t f,
    in_denotational_sem_lev_goal l (proj1_sig (Prog r) t) f →
    in_denotational_sem_lev_goal (S l) (Invoke r t) f.
```

The lemma relating bounded and unbounded denotational semantics is translated into Coq:

```
Lemma in_denotational_sem_some_lev: ∀ (g : goal) (f : repr_fun),
    in_denotational_sem_goal g f →
    ∃ l, in_denotational_sem_lev_goal l g f.
```

The statements of the theorems are as follows:

```
Theorem search_correctness: ∀ (g : goal) (k : nat) (f : repr_fun) (t : trace),
  closed_goal_in_context (first_nats k) g) →
  op_sem (State (Leaf g empty_subst k)) t) →
  in_denotational_analog t f →
  in_denotational_sem_goal g f.

Theorem search_completeness: ∀ (g : goal) (k : nat) (f : repr_fun) (t : trace),
  closed_goal_in_context (first_nats k) g) →
  op_sem (State (Leaf g empty_subst k)) t) →
  in_denotational_sem_goal g f →
  ∃ (f' : repr_fun), (in_denotational_analog t f') ∧
                     ∀ (x : var), In x (first_nats k) → f x = f' x.
```

One important immediate corollary of these theorems is the correctness of certain program transformations. Since the results obtained by the search on a specification are exactly the results from the mathematical model of this specification, after the transformations of relations that do not change their mathematical meaning the search will obtain the same results. Note that this way we guarantee only the stability of results as the set of ground terms, the other aspects of program behavior, such as termination, may be affected. This allows us to safely (to a certain extent) apply such natural transformations as:

- changing the order of constituents in conjunction or disjunction;
- swapping conjunction and disjunction using distributivity;
- moving fresh variable introduction.

and even transform relational definitions to some kinds of normal form (like all fresh variables introduction on the top level with the conjunctive normal form inside), which may be convenient, for example, for metacomputation.

6 CONCLUSION AND FUTURE WORK

In this paper we presented a formal semantics for core miniKanren and proved some its basic properties, which are believed to hold in existing implementations. We consider our work as an initial setup for a future development of miniKanren semantics. The language we considered here lacks many important features, which are already introduced and employed in many implementations. Integrating these extensions — in the first hand, disequality constraints, — into the semantics looks a natural direction for future work. We also are going to address the problems of proving some properties of relational programs (equivalence, refutational completeness, etc.).

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*.

Franz Baader and Wayne Snyder. 2001. Handbook of Automated Reasoning. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, Chapter Unification Theory.

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

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University.

William E. Byrd and Daniel P. Friedman. 2007. αkanren: A Fresh Name in Nominal Logic Programming. In *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming*. 79–90.

William E. Byrd, Daniel P. Friedman, Ramana Kumar, and Joseph P. Near. [n. d.]. A Shallow Scheme Embedding of Bottom-Avoiding Streams. ([n. d.]).

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The 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 and Daniel P. Friedman. 2015. A Framework for Extending microKanren with Constraints. In *Proceedings of the 2015 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. *SIGPLAN Not.* 52, 2 (Nov. 2016), 96–107. https://doi.org/10.1145/3093334.2989230

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). *SIGPLAN Not.* 40, 9 (Sept. 2005), 192–203. https://doi.org/10.1145/1090189.1086390

J. W. Lloyd. 1984. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg.

Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2017. Typed Relational Conversion. In *Proceedings of the International Symposium on Trends in Functional Programming*.

Conor McBride. 2003. First-order Unification by Structural Recursion. *J. Funct. Program.* 13, 6 (Nov. 2003), 1061–1075. https://doi.org/10.1017/S0956796803004957

F. Pfenning and C. Elliott. 1988. Higher-order Abstract Syntax. *SIGPLAN Not.* 23, 7 (June 1988), 199–208. https://doi.org/10.1145/960116.54010

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 '18)*. ACM, New York, NY, USA, Article 18, 13 pages. https://doi.org/10.1145/3236950.3236958

Cameron Swords and Daniel P. Friedman. 2013. rKanren: Guided Search in miniKanren. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*.

Alfred Tarski. 1955. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific J. Math.* 5 (06 1955). https://doi.org/10.2140/pjm.1955.5.285

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146