# Certified Semantics for Relational Programming[⋆]

Dmitry Rozplokhas[1,3][0000-0001-7882-4497], Andrey Vyatkin[2], and Dmitry Boulytchev[2,3][0000-0001-8363-7143]

[1] Higher School of Economics, Russia
[2] Saint Petersburg State University, Russia
[3] JetBrains Research, Russia

**Abstract.** We present a formal study of semantics for the relational programming language miniKanren. First, we formulate a denotational semantics which corresponds to the minimal Herbrand model for definite logic programs. Second, we present operational semantics which models interleaving, the distinctive feature of miniKanren implementation, and prove its soundness and completeness w.r.t. the denotational semantics. Our development is supported by a Coq specification, from which a reference interpreter can be extracted. We also derive from our main result a certified semantics (and a reference interpreter) for SLD resolution with cut and prove its soundness.

## 1 Introduction

In the context of this paper, we understand "relational programming" as a puristic form of logic programming with all extra-logical features banned. Specifically, we use miniKanren as an exemplary language; miniKanren can be seen as a logical language with explicit connectives, existentials and unification, and is mutually convertible to the pure logical subset of Prolog.[1] Unlike Prolog, which relies on SLD-resolution, most miniKanren implementations use a monadic *interleaving search*, which is known to be complete [15]. miniKanren is designed as a shallow DSL which may help to equip the host language with logical reasoning features. This design choice has been proven to be applicable in practice, and there are more than 100 implementations for almost 50 languages.

Although there already were attempts to define a formal semantics for miniKanren, none of them were capable of reflecting the distinctive property of miniKanren's search — *interleaving* [18]. Since this distinctive search strategy is essential for the specification of the language and its extensions, the description of almost all development on miniKanren was not based on formal semantics. The introductory book on miniKanren [12] describes the language by means of an evolving set of examples. In a series of follow-up papers [13,1,14,7,?,30] various extensions of the language were presented with their semantics explained in terms of a Scheme implementation. We argue that this style of semantic definition is fragile and not self-sufficient since it relies on concrete implementation

⋆ The reported study was funded by RFBR, project number 18-01-00380.

[1] A detailed Prolog-to-miniKanren comparison can be found here: `http://minikanren.org/minikanren-and-prolog.html`

| | | |
|---|---|---|
| $\mathcal{C}$ | $= \{C_i^{k_i}\}$ | 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 |
| $\mathcal{G}$ | $= \mathcal{T}_\mathcal{X} \equiv \mathcal{T}_\mathcal{X}$ | unification |
| | $\mathcal{G} \wedge \mathcal{G}$ | 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 the source language

languages' semantics and therefore is not stable under the host language replacement. In addition, the justification of important properties of the language and specific relational programs becomes cumbersome.

In this paper, we present a formal semantics for core miniKanren and prove some of its basic properties. First, we define denotational semantics similar to the least Herbrand model for definite logic programs [23]; then we describe operational semantics with interleaving in terms of a labeled transition system. Finally, we prove soundness and completeness of the operational semantics w.r.t the denotational one. We support our development with a formal specification using the Coq proof assistant [4], thus outsourcing the burden of proof checking to the automatic tool and deriving a certified reference interpreter via the extraction mechanism. As a rather straightforward extension of our main result, we also provide a certified operational semantics (and a reference interpreter) for SLD resolution with cut, a new result to our knowledge; while this step brings us out of purely relational domain, it still can be interesting on its own.

## 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 in Fig. 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 $\mathcal{X}$. The second kind, *semantic* or *logic* variables, is denoted by $\mathcal{A}$. 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 *goal*. In our case, there are 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. The final syntactic category is a *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 a set of free variables for both terms and goals is conventional; 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 run time.

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 that maps semantic variables into the terms over semantic variables. Then five types of scenarios are possible (depending 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 a 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 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 $\alpha$, substitutes all free occurrences of $x$ in $g$ with $\alpha$, and runs the goal.
– Invocation "$R_i^{k_i}\ (t_1,...,t_{k_i})$" finds a definition for the relational symbol $R_i^{k_i} = \lambda x_1 \dots x_{k_i} \,.\, g_i$, substitutes all free occurrences of a 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 that the final substitutions for these variables constitute the result of the goal evaluation.

Conjunction and disjunction form a monadic [32] interface with conjunction playing role of "`bind`" and disjunction the role 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 evaluated incrementally with the control passing from one disjunct to another (*interleaving*) [18]; the evaluation of some goals can be additionally deferred (via so-called "*inverse-$\eta$-delay*") [13]; instead of streams the implementation can be based on "ferns" [8] to defer divergent computations, etc. In the following sections, we present a complete formal description of relational semantics which resolves these uncertainties in a conventional way.

As an example consider the following specification. For the sake of brevity we abbreviate immediately nested "**fresh**" constructs into the one, writing "**fresh** $x\ y \dots .\ g$" instead of "**fresh** $x$ . **fresh** $y \,.\, \dots .\ g$".

```
append^o =λ x y xy .                  revers^o =λ x xr .
 ((x ≡ Nil) ∧ (xy ≡ y)) ∨              ((x ≡ Nil) ∧ (xr ≡ Nil)) ∨
 (fresh h t ty .                       (fresh h t tr .
   (x  ≡ Cons (h, t))   ∧                (x ≡ Cons (h, t)) ∧
   (xy ≡ Cons (h, ty))  ∧                (append^o tr (Cons (h, Nil)) xr) ∧
   (append^o t y ty));                   (revers^o t tr));

revers^o x x
```

Here we defined[2] two relational symbols — "append$^o$" and "revers$^o$", — and specified a top-level goal "revers$^o$ x x". The symbol "append$^o$" defines a relation of concatenation of lists — it takes three arguments and performs a case analysis on the first one. If the first argument 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. palindromes. Running it results in an infinite stream of substitutions:

$\alpha \mapsto$ Nil
$\alpha \mapsto$ Cons ($\beta_0$, Nil)
$\alpha \mapsto$ Cons ($\beta_0$, Cons ($\beta_0$, Nil))
$\alpha \mapsto$ Cons ($\beta_0$, Cons ($\beta_1$, Cons ($\beta_0$, Nil)))
...

where "$\alpha$" is a *semantic* variable, corresponding to "x", "$\beta_i$" are free semantic variables. Therefore, each substitution represents a set of all palindromes of a certain length.

## 3 Denotational Semantics

In this section, we present a denotational semantics for the language we defined above. We use a simple set-theoretic approach analogous to the least Herbrand model for definite logic programs [23]. 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 example, nominal constructs [7]) which we plan to do in future.

[2] We respect here a conventional tradition for miniKanren programming to superscript all relational names with "$^o$".

We also must make the following observations. First, building inductive denotational semantics in a conventional way amounts to constructing a complete lattice and a monotone function and taking its least fixed point [31]. As we deal with a first-order language with only monotonic constructs (conjunction/disjunction) these steps are trivial. Moreover, we express the semantics in Coq, where all well-formed inductive definitions already have proper semantics, which removes the necessity to justify the validity of the steps we perform. Second, the least Herbrand model is traditionally defined as the least fixed point of a transition function (defined by a logic program) which maps sets of ground atoms to sets of ground atoms. We are, however, interested in *relational* semantics which should map a program into $n$-ary relation over ground terms, where $n$ is the number of free variables in the topmost goal. Thus, we deviate from the traditional route and describe the denotational semantics in a more specific way.

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

$\mathtt{x} \equiv \mathtt{Cons\ (y,\ z)}$

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

$\alpha \mapsto\ \mathtt{Cons}\ (\beta,\ \gamma)$

where semantic variables $\alpha$, $\beta$ and $\gamma$ 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 "$\beta$" and "$\gamma$" in the answer. This answer can be seen as the following ternary relation over the set of all ground terms:

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

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

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

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 the 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

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

which maps terms to terms:

$$\begin{array}{c}\bar{\mathfrak{f}}\,(\alpha) = \mathfrak{f}\,(\alpha)\\ \bar{\mathfrak{f}}\,(C_i^{k_i}\,(t_1, \dots t_{k_i})) = C_i^{k_i}\,(\bar{\mathfrak{f}}\,(t_1), \dots \bar{\mathfrak{f}}\,(t_{k_i}))\end{array}$$

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 $\theta$ which turns both $t_1$ and $t_2$ into the same *ground* term (we do not require $\theta$ to be the most general). Thus, $\theta$ maps (some) 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 $\bar{\mathfrak{f}}(t_1) = \bar{\mathfrak{f}}(t_2)$.

We will use the conventional notions of pointwise modification of a function $f\,[x \leftarrow v]$ and substitution $g\,[t/x]$ of a free variable $x$ with a term $t$ in a goal (or a term) $g$.

For a representing function $\mathfrak{f} : \mathcal{A} \to \mathcal{D}$ and a semantic variable $\alpha$ 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 Fig. 2). We've already given the motivation for the semantics of unification: the condition $\bar{\mathfrak{f}}(t_1) = \bar{\mathfrak{f}}(t_2)$ 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. In the case of a relational invocation we unfold the definition of the corresponding relational symbol and substitute its formal parameters with actual ones.

The only non-trivial case is that of "**fresh** $x$ . $g$". First, we take an arbitrary semantic variable $\alpha$, not free in $g$, and substitute $x$ with $\alpha$. 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 $\alpha$ since in our model the semantics of a goal specifies a relation over its free variables. We introduce some nondeterminism by choosing arbitrary $\alpha$, but we can prove that with different choices of free variable the semantics of a goal does not change.

**Lemma 1.** *For any goal **fresh** $x$ . $g$, for any two variables $\alpha$ and $\beta$ which are not free in this goal, if $\mathfrak{f} \in [\![g\,[\alpha/x]]\!]$, then for any representing function $\mathfrak{f}'$, such that*

$$\begin{array}{lll}
[\![t_1 \equiv t_2]\!] & = \{\mathfrak{f} : \mathcal{A} \to \mathcal{D} \mid \bar{\mathfrak{f}}\,(t_1) = \bar{\mathfrak{f}}\,(t_2)\} & [\textsc{Unify}_D] \\
[\![g_1 \wedge g_2]\!] & = [\![g_1]\!] \cap [\![g_2]\!] & [\textsc{Conj}_D] \\
[\![g_1 \vee g_2]\!] & = [\![g_1]\!] \cup [\![g_2]\!] & [\textsc{Disj}_D] \\
[\![\,\mathbf{fresh}\ x\,.\,g]\!] & = ([\![g\,[\alpha/x]]\!]) \uparrow \alpha,\ \alpha \notin FV(g) & [\textsc{Fresh}_D] \\
[\![R\,(t_1,\dots,t_k)]\!] & = [\![g\,[t_1/x_1,\dots,t_k/x_k]]\!],\ \text{where } R = \lambda\,x_1\dots x_k\,.\,g & [\textsc{Invoke}_D]
\end{array}$$

**Fig. 2.** Denotational semantics of goals

*1.* $\mathfrak{f}'(\beta) = \mathfrak{f}(\alpha)$
*2.* $\forall \gamma : \gamma \neq \alpha \wedge \gamma \neq \beta,\ \mathfrak{f}'(\gamma) = \mathfrak{f}(\gamma)$
*it is true that* $\mathfrak{f}' \in [\![g\,[\beta/x]]\!]$.

The proof turned out to be the most cumbersome among all others in the case where $g$ is a nested **fresh** contruct. In that case, we have to constructively build two representing functions (including an intermediate one for an intermediate goal) by pointwise modification. The details of this proof can be found in the extended version of the paper.[3]

We can prove the following important *closedness condition* for the semantics of a goal $g$.

**Lemma 2 (Closedness condition).** *For any goal $g$ and two representing functions $\mathfrak{f}$ and $\mathfrak{f}'$, such that $\mathfrak{f}|_{FV(g)} = \mathfrak{f}'|_{FV(g)}$, it is true, that $\mathfrak{f} \in [\![g]\!] \Leftrightarrow \mathfrak{f}' \in [\![g]\!]$.*

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 closed in the sense that it does not introduce artificial restrictions for the relation it defines.

## 4 Operational Semantics

In this section we describe the operational semantics of miniKanren, which corresponds to the known implementations with interleaving search. The semantics is given in the form of a labeled transition system (LTS) [17]. 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 $\Sigma$ the set of all substitutions, $\mathcal{D}om\,(\sigma)$ — the domain for a substitution $\sigma$, $\mathcal{VR}an\,(\sigma) = \bigcup_{\alpha \in \mathcal{D}om\,(\sigma)} \mathcal{FV}\,(\sigma\,(\alpha))$ — its range (the set of all free variables in the image).

The *non-terminal 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

[3] The extended version of this paper is available at `https://arxiv.org/abs/2005.01018`

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 $g$ is a goal, $\sigma$ is a substitution accumulated so far, and $n$ is 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.

The full set of states also include one separate terminal state (denoted by $\diamond$), which symbolizes the end of the evaluation.

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

We will operate with the well-formed states only, which are defined as follows.

**Definition 1.** *Well-formedness condition for extended states:*
- $\diamond$ *is well-formed;*
- $\langle g, \sigma, n\rangle$ *is well-formed iff* $\mathcal{FV}(g) \cup \mathcal{D}om(\sigma) \cup \mathcal{VR}an(\sigma) \subseteq \{\alpha_1, \dots, \alpha_n\}$*;*
- $s_1 \oplus s_2$ *is well-formed iff* $s_1$ *and* $s_2$ *are well-formed;*
- $s \otimes g$ *is well-formed iff* $s$ *is well-formed and for all leaf triplets* $\langle \_, \_, n\rangle$ *in* $s$ *it is true that* $\mathcal{FV}(g) \subseteq \{\alpha_1, \dots, \alpha_n\}$*.*

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.

The transition rules are shown in Fig. 3. The first two rules specify the semantics of unification. If two terms are not unifiable under the current substitution $\sigma$ then the evaluation stops with no answer; otherwise, it stops with the most general unifier applied to a current substitution as an answer.

The next two rules describe the steps performed when disjunction or 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.

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, the disjunction evaluation proceeds with the right state, propagating the label (SumStop and SumStep),

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Fig. 3.** Operational semantics of interleaving search

and the conjunction schedules the right goal for evaluation in the context of the returned answer (ProdStopAns) or stops if there is no answer (ProdStop).

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 (SumStep, SumStepAns). In the 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 (ProdStep); 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 (ProdStepAns).

The introduced transition system is completely deterministic: there is exactly one transition from any non-terminal state. There is, 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. This choice reflects the inherent non-deterministic nature of search in relational (and, more generally, logical) programming. Although we could introduce this ambiguity into the semantics (by replacing specific rules for disjunctions and conjunctions evaluation with some conditions on it), we want an operational semantics that would be easy to present and easy to employ to describe existing language extensions (already described for a specific implementation of interleaving search), so we instead base the semantics on one canonical search strategy. At the same time, as long as deterministic search procedures are sound and complete, we can consider them "equivalent".[4]

It is easy to prove that transitions preserve well-formedness of states.

**Lemma 3.** *(Well-formedness preservation) For any transition $s \xrightarrow{l} \hat{s}$, if $s$ is well-formed then $\hat{s}$ is also well-formed.*

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

We can relate sets of answers for the partially evaluated conjunction and disjunction with sets of answers for their constituents by the two following lemmas.

**Lemma 4.** *For any non-terminal states $s_1$ and $s_2$, $\mathcal{T}r_{s_1 \oplus s_2} = \mathcal{T}r_{s_1} \cup \mathcal{T}r_{s_2}$.*

**Lemma 5.** *For any non-terminal state $s$ and goal $g$, $\mathcal{T}r_{s \otimes g} \supseteq \bigcup_{(\sigma,n) \in \mathcal{T}r_s} \mathcal{T}r_{\langle g,\sigma,n\rangle}$.*

These two lemmas constitute the exact conditions on definition of these operators that we will use to prove the completeness of an operational semantics.

We also can easily describe the criterion of termination for disjunctions.

**Lemma 6.** *For any goals $g_1$ and $g_2$, sunbstitution $\sigma$, and number $n$, the trace from the state $\langle g_1 \vee g_2, \sigma, n\rangle$ is finite iff the traces from both $\langle g_1, \sigma, n\rangle$ and $\langle g_2, \sigma, n\rangle$ are finite.*

These simple statements already allow us to prove two important properties of interleaving search as corollaries: the "fairness" of disjunction — the fact that the trace for disjunction contains all the answers from both streams for disjuncts — and the "commutativity" of disjunctions — the fact that swapping two disjuncts (at the top level) does not change the termination of the goal evaluation.

## 5 Equivalence of Semantics

Now we can relate two different kinds of semantics for miniKanren described in the previous sections and show that the results given by these two semantics are the same for any specification. This will actually say something important

[4] There still can be differences in observable behavior of concrete goals under different sound and complete search strategies. For example, a goal can be refutationally complete [6] under one strategy and non-complete under another.

$$\begin{aligned}
[\![\Diamond]\!] &= \varnothing \\
[\![\langle g, \sigma, n\rangle]\!] &= [\![g]\!] \cap [\![\sigma]\!] \\
[\![s_1 \oplus s_2]\!] &= [\![s_1]\!] \cup [\![s_2]\!] \\
[\![s \otimes g]\!] &= [\![s]\!] \cap [\![g]\!]
\end{aligned}$$

**Fig. 4.** Denotational semantics of states

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 a mathematical point of view, by proving their equivalence we establish the *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 the numbers of allocated variables) for the operational one and a set of representing functions for the denotational one. We can notice that the notion of a representing function is close to substitution, with only two differences:

– representing functions are total;
– terms in the domain of representing functions are ground.

Therefore we can easily extend (perhaps ambiguously) any substitution to a representing function by composing it with an arbitrary representing function preserving all variable dependencies in the substitution. So we can define a set of representing functions that correspond to a substitution as follows:

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

And the *denotational analog* of operational semantics (a set of representing functions corresponding to the answers in the trace) for a given state $\hat{s}$ is then defined as the union of sets for all substitutions in the trace:

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

This allows us to state theorems relating the two semantics.

**Theorem 1 (Operational semantics soundness).** *If indices of all free variables in a goal $g$ are limited by some number $n$, then $[\![\langle g, \epsilon, n\rangle]\!]_{op} \subseteq [\![g]\!]$.*

It can be proven by nested induction, but first, we need to generalize the statement so that the inductive hypothesis is strong enough for the inductive step. To do so, we define denotational semantics not only for goals but for arbitrary states. Note that this definition does not need to have any intuitive interpretation, it is introduced only for the proof to go smoothly. The definition of the denotational semantics for extended states is shown on Fig. 4. The generalized version of the theorem uses it.

**Lemma 7 (Generalized soundness).** *For any well-formed state $\hat{s}$*

$$[\![\hat{s}]\!]_{op} \subseteq [\![\hat{s}]\!].$$

It can be proven by the induction on the number of steps in which a given answer (more accurately, the substitution that contains it) occurs in the trace. We break the proof in two parts and separately prove by induction on evidence that for every transition in our system the semantics of both the label (if there is one) and the next state are subsets of the denotational semantics for the initial state.

**Lemma 8 (Soundness of the answer).** *For any transition* $s \xrightarrow{(\sigma,n)} \hat{s}$*,* $[\![\sigma]\!] \subseteq [\![s]\!]$*.*

**Lemma 9 (Soundness of the next state).** *For any transition* $s \xrightarrow{l} \hat{s}$*,* $[\![\hat{s}]\!] \subseteq [\![s]\!]$*.*

It would be tempting to formulate the completeness of operational semantics as soundness with the inverted inclusion, but it does not hold in such generality. The reason for this is that the denotational semantics encodes only the dependencies between free variables of a goal, which is reflected by the closedness condition, while the operational semantics may also contain dependencies between semantic variables allocated in **fresh** constructs. Therefore we formulate 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 returns substitutions only for queried variables, which are allocated in the beginning.

**Theorem 2 (Operational semantics completeness).** *If the indices of all free variables in a goal* $g$ *are limited by some number* $n$*, then*

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

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 — *step-indexed denotational semantics* (denoted by $[\![\bullet]\!]^i$). It is an implementation of the well-known approach [2] of indexing typing or semantic logical relations by a number of permitted evaluation steps to allow inductive reasoning on it. In our case, $[\![g]\!]^i$ includes only those representing functions that one can get after no more than $i$ unfoldings of relational calls.

The step-indexed denotational semantics is an approximation of the conventional denotational semantics; it is clear that any answer in conventional denotational semantics will also be in step-indexed denotational semantics for some number of steps.

**Lemma 10.** $[\![g]\!] \subseteq \cup_i [\![g]\!]^i$

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

**Lemma 11 (Generalized completeness).** *For any set of relational definitions, for any number of unfoldings $i$, for any well-formed state $\langle g, \sigma, n\rangle$,*

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

The proof is by the induction on nuber of unfoldings $i$. The induction step is proven by structural induction on goal $g$. We use lemmas 4 and 5 for evaluation of a disjunction and a conjunction respectively, and lemma 1 in the case of fresh variable introduction to move from an arbitrary semantic variable in denotational semantics to the next allocated fresh variable. The details of this proof may be found in the extended version of the paper.

## 6 Specification in Coq

We certified all the definitions and propositions from the previous sections using the Coq proof assistant.[5] The Coq specification for the most part closely follows the formal descriptions we gave by means of inductive definitions (and inductively defined propositions in particular) and structural induction in proofs. The detailed description of the specification, including code snippets, is provided in the extended version of the paper, and in this section we address only some non-trivial parts of it and some design choices.

The language formalized in Coq has a few non-essential simplifications for the sake of convenience. Specifically, we restrict the arities of all constructors to be either zero or two and require all relations to have exactly one argument. These restrictions do not make the language less expressive in any way since we can always represent a sequence of terms as a list using constructors $\texttt{Nil}^0$ and $\texttt{Cons}^2$.

In our formalization of the language we use higher-order abstract syntax [27] for variable binding, therefore we work explicitly only with semantic variables. We preferred it to the first-order syntax because it gives us the ability to use substitution and the induction principle provided by Coq. On the other hand, we need to explicitly specify a requirement on the syntax representation, which is trivially fulfilled in the first-order case: all bindings have to be "consistent", i.e. if we instantiate a higher-order **fresh** construct with different semantic variables the results will be the same up to some renaming (provided that both those variables are not free in the body of the binder). Another requirement we have to specify explicitly (independent of HOAS/FOAS dichotomy) is a requirement that the definitions of relations do not contain unbound semantic variables.

To formalize the operational semantics in Coq we first need to define all preliminary notions from unification theory [3] which our semantics uses. In particular, we need to implement the notion of the most general unifier (MGU). As it is well-known [25] all standard recursive algorithms for calculating MGU are not decreasing on argument terms, so we can't define them as simple recursive functions in Coq due to the termination check failure. The standard approach to tackle this problem is to define the function through well-founded recursion.

[5] The specification is available at `https://github.com/dboulytchev/miniKanren-coq`

We use a distinctive version of this approach, which is more convenient for our purposes: we define MGU as a proposition (for which there is no termination requirement in Coq) with a dedicated structurally-recursive function for one step of unification, and then we use a well-founded induction to prove the existence of a corresponding result for any arguments and defining properties of MGU. For this well-founded induction, we use the number of distinct free variables in argument terms as a well-founded order on pairs of terms.

In the operational semantics, to define traces as (possibly) infinite sequences of transitions we use the standard approach in Coq — coinductively defined streams. Operating with them requires a number of well-known tricks, described by Chlipala [9], to be applied, such as the use of a separate coinductive definition of equality on streams.

The final proofs of soundness and completeness of operational semantics are relatively small, but the large amount of work is hidden in the proofs of auxiliary facts that they use (including lemmas from the previous sections and some technical machinery for handling representing functions).

## 7 Applications

In this section, we consider some applications of the framework and results, described in the previous sections.

### 7.1 Correctness of Transformations

One important immediate corollary of the equivalence theorems we have proven is the justification of correctness for certain program transformations. The completeness of interleaving search guarantees the correctness of any transformation that preserves the denotational semantics, for example:

- changing the order of constituents in conjunctions and disjunctions;
- distributing conjunctions over disjunctions and vice versa, for example, normalizing goals info CNF or DNF;
- moving fresh variable introduction upwards/downwards, for example, transforming any relation into a top-level fresh construct with a freshless body.

Note that this way we can guarantee only the preservation of results as *sets of ground terms*; the other aspects of program behavior, such as termination, may be affected by some of these transformations.[6]

One of the applications for these transformations is a conversion from/to Prolog. As both languages use essentially the same fragment of first-order logic, their programs are mutually convertible. The conversion from Prolog to miniKanren is simpler as the latter admits a richer syntax of goals. The inverse conversion involves the transformation into a DNF and splitting the disjunction into a number of separate clauses. This transformation, in particular, makes it possible to reuse our approach to describe the semantics of Prolog as well. In the following sections we briefly address this problem.

[6] Possible slowdown and loss of termination after reorderings in conjunction is a famous example of this phenomenon in miniKanren, known as conjunction non-commutativity [6].

### 7.2 SLD Semantics

The conventional Prolog SLD search differs from the interleaving one in just one aspect — it does not perform interleaving. Thus, changing just two rules in the operational semantics converts interleaving search into the depth-first one:

$$\frac{s_1 \xrightarrow{\circ} s'_1}{(s_1 \oplus s_2) \xrightarrow{\circ} (s'_1 \oplus s_2)}\ [\textsc{DisjStep}] \qquad \frac{s_1 \xrightarrow{r} s'_1}{(s_1 \oplus s_2) \xrightarrow{r} (s'_1 \oplus s_2)}\ [\textsc{DisjStepAns}]$$

With this definition we can almost completely reuse the mechanized proof of soundness (with minor changes); the completeness, however, can no longer be proven (as it does not hold anymore).

### 7.3 Cut

Dealing with the "cut" construct is known to be a cornerstone feature in the study of operational semantics for Prolog. It turned out that in our case the semantics of "cut" can be expressed naturally (but a bit verbosely). Unlike SLD-resolution, it does not amount to an incremental change in semantics description. It also would work only for programs directly converted from Prolog specifications.

The key observation in dealing with the "cut" in our setting is that a state in our semantics, in fact, encodes the whole current search tree (including all backtracking possibilities). This opens the opportunity to organize proper "navigation" through the tree to reflect the effect of "cut". The details of the semantic description can be found in the extended version of the paper.

For this semantics, we can repeat the proof of soundness w.r.t. to the denotational semantics. There is, however, a little subtlety with our construction: we cannot formally prove that our semantics indeed encodes the conventional meaning of "cut" (since we do not have other semantics of "cut" to compare with). Nevertheless, we can demonstrate a plausible behavior using the extracted reference interpreter.

### 7.4 Reference Interpreters

Using the Coq extraction mechanism, we extracted two reference interpreters from our definitions and theorems: one for conventional miniKanren with interleaving search and another one for SLD search with cut. These interpreters can be used to practically investigate the behavior of specifications in unclear, complex, or corner cases. Our experience has shown that these interpreters demonstrate the expected behavior in all cases.

## 8 Related Work

The study of formal semantics for logic programming languages, particularly Prolog, is a well-established research domain. Early works [16,10] addressed the computational aspects of both pure Prolog and its extension with the cut construct. Recently, the application of certified/mechanized approaches came into focus as well. In particular, in one work [21] the equivalence of a few differently defined semantics for pure Prolog is proven, and in another work [20] a denotational semantics for Prolog with cut is presented; both works provide Coq-mechanized proofs. It is interesting that the former one also advocates the use of higher-order abstract syntax. We are not aware of any prior work on certified semantics for Prolog which contributed a correct-by-construction interpreter. Our certified description of SLD resolution with cut can be considered as a certified semantics for Prolog modulo occurs check in unification (which Prolog does not have by default).

The implementation of first-order unification in dependently typed languages constitutes a well-known challenge with a number of known solutions. The major difficulty comes from the non-structural recursivity of conventional unification algorithms, which requires to provide a witness for convergence. The standard approach is to define a generally-recursive function and a well-founded order for its arguments. This route is taken in a number of works [26,5,19,28], where the descriptions of unification algorithms are given in Coq, LCF and Alf. The well-founded used there is lexicographically ordered tuples, containing the information about the number of different free variables and the sizes of the arguments. We implement a similar approach, but we separate the test for the non-matching case into a dedicated function. Thus, we make a recursive call only when the current substitution extension is guaranteed, which allows us to use the number of different free variables as the well-founded order. An alternative approach suggested by McBride [25] gives a structurally recursive definition of the unification algorithm; this is achieved by indexing the arguments with the numbers of their free variables.

The use of higher-order abstract syntax (HOAS) for dealing with language constructs in Coq was addressed in early work [11], where it was employed to describe the lambda calculus. The inconsistency phenomenon of HOAS representation, mentioned in Section 6, is called there "exotic terms" there and is handled using a dedicated inductive predicate "`Valid_v`". The predicate has a non-trivial implementation based on subtle observations on the behavior of bindings. Our case, however, is much simpler: there is not much variety in "exotic terms" (for example, we do not have reductions in terms), and our consistency predicate can be considered as a limited version of "`Valid_v`" for a smaller language.

The study of formal semantics for miniKanren is not a completely novel venture. Previously, a nondeterministic small-step semantics was described [24], as well as a big-step semantics for a finite number of answers [29]; neither uses proof mechanization and in both works the interleaving is not addressed.

The work of Kumar [22] can be considered as our direct predecessor. It also introduces both denotational and operational semantics and presents a HOL-certified proof for the soundness of the latter w.r.t. the former. The denotational semantics resembles ours but considers only queries with a single free variable (we do not see this restriction as important). On the other hand, the operational semantics is non-deterministic, which makes it impossible to express interleaving and extract the interpreter in a direct way. In addition, a specific form of "executable semantics" is introduced, but its connection to the other two is not established. Finally, no completeness result is presented. We consider our completeness proof as an essential improvement.

The most important property of interleaving search — completeness — was postulated in the introductory paper [18], and is delivered by all major implementations. Hemann et al. [15] give a proof of completeness for a specific implementation of MINIKANREN; however, the completeness is understood there as preservation of all answers during the interleaving of answer streams, i.e. in a more narrow sense than in our work since no relation to denotational semantics is established.

## 9 Conclusion and Future Work

In this paper, we presented a certified formal semantics for core MINIKANREN and proved some of its basic properties (including interleaving search completeness, disjunction fairness and commutativity), which are believed to hold in existing implementations. We also derived a semantics for conventional SLD resolution with cut and extracted two certified reference interpreters. We consider our work as the initial setup for the 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 are also going to address the problems of proving some properties of relational programs (equivalence, refutational completeness, etc.).

## References

1. C. E. Alvis, J. J. Willcock, K. M. Carter, W. E. Byrd, and D. P. Friedman. cKanren: miniKanren with constraints. In *Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming*, Oct. 2011.
2. A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.
3. F. Baader and W. Snyder. Handbook of automated reasoning. chapter Unification Theory. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
5. A. Bove. Programming in martin-löf type theory: Unification – a non-trivial example. In *DEPARTMENT OF COMPUTER SCIENCE, CHALMERS UNIVERSITY OF TECHNOLOGY*, pages 22–42, 1999.
6. W. E. Byrd. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis, Indiana University, September 2009.
7. W. E. Byrd and D. P. Friedman. $\alpha$kanren: A fresh name in nominal logic programming. In *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming*, pages 79–90, 2007.

8. W. E. Byrd, D. P. Friedman, R. Kumar, and J. P. Near. A shallow Scheme embedding of bottom-avoiding streams. In *To appear in a special issue of Higher-Order and Symbolic Computation, in honor of Mitchell Wand's 60th birthday*.
9. A. Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.
10. S. K. Debray and P. Mishra. Denotational and operational semantics for PROLOG. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, pages 245–274, 1987.
11. J. Despeyroux, A. P. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, pages 124–138, 1995.
12. D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The reasoned schemer*. MIT Press, 2005.
13. J. Hemann and D. P. Friedman. $\mu$Kanren: A minimal functional core for relational programming. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*, 2013.
14. J. Hemann and D. P. Friedman. A framework for extending microKanren with constraints. In *Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, WLP 2015 / WLP 2016 / WFLP 2016, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016*, pages 135–149, 2017.
15. J. Hemann, D. P. Friedman, W. E. Byrd, and M. Might. 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*, pages 96–107, 2016.
16. N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA, February 6-9, 1984*, pages 281–288, 1984.
17. R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.
18. O. Kiselyov, C. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). pages 192–203, 2005.
19. S. Kothari and J. Caldwell. A machine checked model of idempotent MGU axioms for lists of equational constraints. In *Proceedings 24th International Workshop on Unification, UNIF 2010, Edinburgh, United Kingdom, 14th July 2010*, pages 24–38, 2010.
20. J. Kriener and A. King. Semantics for Prolog with cut - revisited. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 270–284, 2014.
21. J. Kriener, A. King, and S. Blazy. Proofs you can believe in: proving equivalences between Prolog semantics in Coq. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 37–48, 2013.
22. R. Kumar. Mechanising aspects of miniKanren in HOL. Bachelor Thesis, The Australian National University, 2010.
23. J. W. Lloyd. *Foundations of Logic Programming, 1st Edition*. Springer, 1984.

24. P. Lozov, A. Vyatkin, and D. Boulytchev. Typed relational conversion. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*, pages 39–58, 2017.
25. C. McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13(6):1061–1075, 2003.
26. L. C. Paulson. Verifying the unification algorithm in LCF. *Sci. Comput. Program.*, 5(2):143–169, 1985.
27. F. Pfenning and C. Elliott. Higher-order abstract syntax. pages 199–208, 1988.
28. R. G. Ribeiro and C. Camarão. A mechanized textbook proof of a type unification algorithm. In *Formal Methods: Foundations and Applications - 18th Brazilian Symposium, SBMF 2015, Belo Horizonte, Brazil, September 21-22, 2015, Proceedings*, pages 127–141, 2015.
29. D. Rozplokhas and D. Boulytchev. 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*, pages 18:1–18:13, 2018.
30. C. Swords and D. P. Friedman. rKanren: Guided search in miniKanren. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*, 2013.
31. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 06 1955.
32. P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, pages 24–52, 1995.

## A Details of Proofs

### A.1 Proof of Lemma 1

The proof goes by structural induction, all cases naturally use inductive hypotheses, except for the case of fresh variable introduction. We examine this case in more detail.

We have a nested fresh variable introduction **fresh** $y \,.\, g'$ as the goal $g$. $g'$ may contain syntactic variables $x$ and $y$ (the case when $g$ does not contain syntactic variable $x$ is trivial and we have to consider it separately). By the statement of the lemma and the definition of denotational semantics we have $\mathfrak{f}'_1 \in [\![g'\,[\alpha_1/x]\,[\alpha_3/y]]\!]$, where $\alpha_3$ is some fresh semantic variable, and $\mathfrak{f}'_1$ is some representing function which may differ from $\mathfrak{f}_1$ only on variable $\alpha_3$. Variable $\alpha_3$ may not be equal to $\alpha_1$, but may be equal to $\alpha_2$ and further in proof we sometimes have to consider these two cases separetly, because they are essentially different. We should find some fresh variable $\alpha_4$ and representing function $\mathfrak{f}'_2 \in [\![g'\,[\alpha_2/x]\,[\alpha_4/y]]\!]$ that may differ from $\mathfrak{f}_2$ only on variable $\alpha_4$.

Among different options we considered, the easiest choice for a variable $\alpha_4$ turned out to be the freshest possible variable (one that is not equal to any of the mentioned above and that does not occur in the goal $g'$). Then we need to move from the function $\mathfrak{f}'_1$ to the function $\mathfrak{f}'_2$ by applying the induction hypothesis twice (since now we change two substituted variables simultaneously). To do this we need to carefully change the values of the function on a few suitable points. It turned out, that we have to change the values on all four mentioned variables. First, we build an intermediate function $\mathfrak{f}'_{1.5} = \mathfrak{f}'_1[\alpha_3 \leftarrow \mathfrak{f}_2(\alpha_3)][\alpha_4 \leftarrow \mathfrak{f}'_1(\alpha_3)]$ that should belong to an intermediate semantics $[\![g'\,[\alpha_1/x]\,[\alpha_4/y]]\!]$. Then we build a final function $\mathfrak{f}'_2 = \mathfrak{f}'_{1.5}[\alpha_1 \leftarrow \mathfrak{f}_2(\alpha_1)][\alpha_2 \leftarrow \mathfrak{f}'_{1.5}(\alpha_1)]$ on top of it. For both transitions, we carefully check that all conditions on two functions from the statement of the lemma (and, therefore, from the induction hypothesis) are satisfied by examination of values of the functions on all mentioned variables.

### A.2 Proof of Generalized Completeness

The proof goes by induction on the number of unfoldings in step-indexed denotational semantics, then by structural induction on the goal.

- In case of unification $t_1 \equiv t_2$ we need to show that any representing function that unifies terms $t_1$ and $t_2$ and is an extension of $\sigma$ also is an extension of $mgu\,(t_1\sigma, t_2\sigma) \circ \sigma$. This requires some observations about representing functions, most importantly the fact that for any representing function $f$, unifying two terms, there exists a unifying substitution for these terms, for which $f$ is an extension.
- In case of disjunction we apply lemma 4 to induction hypotheses.
- In case of conjunction we apply lemma 5 to induction hypotheses. Note that we use two nested inductions instead of one induction on evidence because of this case: we need to apply an induction hypothesis to different representing functions (which may have different values on non-allocated free variables), and otherwise the induction hypotheses would not be flexible enough.

– In the case of fresh variable introduction, we can not simply use induction hypothesis, because it has a goal with an arbitrary fresh variable substituted (see definition of denotational semantics on Fig. 2), but we need to relate two semantics of the goal with exactly the first non-allocated variable substituted (from transition in operational semantics, see Fig. 3). To overcome this inconsistency, we apply lemma 1 about changing substituted fresh variable. When we do this, we lose the equality of values of representing functions on the first non-allocated variable. It is exactly the place where the proof of the strong version of the completeness theorem (without the restriction of domains of representing functions on allocated variables) fails (as explained in the section 5, the strong version of the completeness theorem does not hold).
– In case of relational invocation we simply use the induction hypothesis with fewer unfoldings.

## B Coq Specification

The Coq specification consists of three parts: specification of preliminary notions that we use in the semantics, specification of the language and semantics for interleaving search, specification of the language extended with cut and semantics for SLD resolution with cut (which repeats the previous part with only a few modifications).

Preliminary notions are notions from unification theory and definition of streams.

From unification theory we need (finitary) terms, substitutions, and operations on them (application to a term, composition). The most important notion we have to formalize here is the most general unifier. As it was described in the section 6, we define it as an inductive relation.

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

It uses an auxiliary function that finds the leftmost mismatch of the two terms (if there is any).

Then we prove by well-founded induction that this relation is a function and it satisfies the defining properties of MGU:

```
Lemma mgu_result_exists : ∀ t1 t2, {r & mgu t1 t2 r}.
Lemma mgu_result_unique : ∀ t1 t2 r r',
    mgu t1 t2 r → mgu t1 t2 r' → r = 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 m,
    mgu t1 t2 (Some m) →
    ∀ (s : subst), unifier s t1 t2 → more_general m s.
Lemma mgu_non_unifiable : ∀ t1 t2,
    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.
```

Possibly infinite streams is a coinductive data type:

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

However, some of its properties we are working with make sense only when defined inductively:

```
Inductive  in_stream : A → stream → Prop := ...
Inductive  finite : stream → Prop := ...
```

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 [9]:

```
CoInductive equal_streams : stream → stream → Prop :=
| eqsNil  :  equal_streams Nil Nil
| eqsCons : ∀ h1 h2 t1 t2,
                h1 = h2 →
                equal_streams t1 t2 →
                equal_streams (Cons h1 t1) (Cons h2 t2).
```

Here we also define a relation for one-by-one interleaving of streams which we will use to describe the evaluation of dijunstions in the operational semantics:

```
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).
```

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

```
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).
```

The syntax of the language can be formalized in Coq in a natural way via inductive data types. Thanks to the higher-order syntax we need to work explicitly only with semantic variables. We use type "name" for all named entities (variables, constructors, and relations), and we define it simply as natural numbers.

As it was noted in the section 6 all terms in our specification have arities either zero or two:

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

And all relations have exactly one argument:

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

We introduce one additional auxilary 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  : (name → goal) → goal
| Invoke : name → term → goal.
```

Note the use of HOAS for fresh variable introduction.

We define sets of free variables for terms naturally as Coq sets:

```
Fixpoint fv_term (t : term) : set name :=
  ...
```

And we use them to define ground terms as a subset type:

```
Definition ground_term : Set :=
  {t : term | fv_term t = empty_set name}.
```

However, for goals it was more convenient to define a set of free variables as a proposition:

```
Inductive is_fv_of_goal (n : name) : goal → Prop :=
  ...
```

We use it to state our two explicit requirements on the syntax representation: the consistency of bindings and the absence of unbound variables in the definitions of relations. The second one is easy to describe:

```
Definition  closed_goal_in_context
  (c : list name) (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).
```

The consistency is based on variable renaming. It turned out to be rather non-trivial to define regular variable renaming for goals in higher-order syntax, but for our purposes a weaker version, which deals only with non-free variables, is sufficient:

```
Inductive  renaming (old_x : name) (new_x : name) :
    goal → goal → Prop :=
...
| rFreshNFV : ∀ fg,
             (~is_fv_of_goal old_x (Fresh fg)) →
             renaming old_x new_x (Fresh fg) (Fresh fg)
| rFreshFV : ∀ fg rfg,
            (is_fv_of_goal old_x (Fresh fg)) →
            (∀ y, (~is_fv_of_goal y (Fresh fg)) →
                 renaming old_x new_x (fg y) (rfg y)) →
            renaming old_x new_x (Fresh fg) (Fresh rfg)
...
```

The consistency for goals and relations then can be defined as follows:

```
Definition  consistent_binding (b : name → goal) : Prop :=
  ∀ x y, (~ is_fv_of_goal x (Fresh b)) →
        renaming x y (b x) (b y).
Inductive  consistent_goal : goal → Prop :=
  ...
Definition  consistent_function
  (f : term → goal) : Prop :=
  ∀ a1 a2 t, renaming a1 a2 (f t)
                 (f (apply_subst [( a1, Var a2)] t)).
Definition  consistent_rel (r : rel) : Prop :=
  ∀ (arg : term), consistent_goal (r arg) ∧
                 consistent_function r.
```

In the snippet above the "`consistent_goal`" property inductively ensures that all bindings occurring in the goal are consistent and "`apply_subst [( a1, Var a2)] t`" in "`consistent_function`" definition renames a variable `a1` into in `a2` term `t`.

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

```
Definition def : Set :=
  {r : rel | closed_rel r ∧ consistent_rel r}.
Definition env : Set := name → def.
Axiom Prog : env.
```

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

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

We define the semantics via the inductive proposition "`in_denotational_sem_goal`" (with notation "[| • , • |]") such that

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

The head of the definition is as follows:

```
Reserved Notation "[| g , f |]" (at level 0).
Inductive in_denotational_sem_goal :
  goal → repr_fun → Prop :=
  ...
where "[| g , f |]" := (in_denotational_sem_goal g f).
```

and the body just goes through the cases shown in Fig. 2.

We also need to explicitly define the step-indexed version of denotational semantics described in the section 5:

```
Reserved Notation "[| n | g , f |]" (at level 0).
Inductive in_denotational_sem_lev_goal :
  nat → goal → repr_fun → Prop :=
  ...
| dslgInvoke : ∀ l r t f,
    [| l | proj1_sig (Prog r) t , f |] →
    [| S l | Invoke r t , f |]
where "[| n | g , f |]" :=
  (in_denotational_sem_lev_goal n g f).
```

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 is closed and consistent. So "`(proj1_sig (Prog r) t)`" here simply takes the body of the corresponding relation.

The lemma relating step-indexed and conventional denotational semantics in Coq looks as follows:

```
Lemma in_denotational_sem_some_lev:
  ∀ (g : goal) (f : repr_fun),
    [| g , f |] → ∃ l, [| l | g , f |].
```

We prove two important properties of the denotational semantics: lemma 1 that states that the choice of a subsituted fresh variable doesn't matter, and the closedness condition (lemma 2). The second proof is by a straightforward structural induction, the structure of the first one is described in the section 10.1.

For operational semantics, the described transition relation can be encoded naturally as an inductively defined proposition (here "`nt_state`" stands for a non-terminal state and "`state`" — for arbitrary one):

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

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

```
Lemma eval_step_exists : ∀ (nst : nt_state),
  {l : label & {st : state & eval_step nst l st}}.
Lemma eval_step_unique : ∀ (nst : nt_state),
    (l1 l2 : label)
    (st1 st2 : state'),
  eval_step nst l1 st1 →
  eval_step nst l2 st2 →
  l1 = l2 ∧ st1 = st2.
```

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 : ∀ nst l st t,
    eval_step nst l st →
    op_sem st t →
    op_sem (State nst) (Cons l t).
Lemma op_sem_exists (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.
```

And we prove the important properties of operational semantics (lemmas 3, 4, 5 and 6).

Finally, we prove both soundness and completeness of the operational semantics of the interleaving search w.r.t. the denotational one. 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 →
    {| t , f |} →
    [| g , f |].
Theorem search_completeness:
  ∀ (g : goal) (k : nat) (f : repr_fun) (t : trace),
    consistent_goal g →
    closed_goal_in_context (first_nats k) g →
    op_sem (State (Leaf g empty_subst k)) t →
    [| g , f |] →
    ∃ (f' : repr_fun),
      {| t , f' |} ∧
      ∀ (x : var), In x (first_nats k) → f x = f' x.
```

Note that we need the consistency requirement only for completeness, not for soundness.

The proof of soundness is by a straightforward structural induction, it is divided between lemmas 8 and 9. The proof of completeness is described in the section 10.2.

The specification of semantics for SLD resolution with cut simply repeats the just described specification of semantics for interleaving search with minimal modification. Specifically, we introduce one additional type of goal (`Cut`), in denotational semantics cuts correspond to the whole domain of representing functions, in operational semantics we distinguish two kinds of `Sum` state, introduce possible cut signal into the definition of transitions and add rules described in the section 12 to the transition system.

```
Inductive cutting_mark : Set :=
| StopCutting : cutting_mark
| KeepCutting : cutting_mark.
Inductive nt_state : Set :=
...
| Sum  : cutting_mark → nt_state → nt_state → nt_state
...
Inductive cut_signal : Set :=
| NoCutting  : cut_signal
| YesCutting : cut_signal.
Inductive eval_step_SLD :
   nt_state → label → cut_signal → state → Set := ...
```

We then repeat the soundness proof for these semantics with minor changes (and skip the completeness proof, since it does not hold for SLD resolution).

$$\frac{s_1 \xrightarrow{\circ} \Diamond}{(s_1 \circledast s_2) \xrightarrow{\circ} s_2}\ [\textsc{AstStop}]$$

$$\frac{s_1 \xrightarrow{r} \Diamond}{(s_1 \circledast s_2) \xrightarrow{r} s_2}\ [\textsc{AstStopAns}]$$

$$\frac{s_1 \xrightarrow{\circ} s_1'}{(s_1 \circledast s_2) \xrightarrow{\circ} (s_2 \circledast s_1')}\ [\textsc{AstStep}]$$

$$\frac{s_1 \xrightarrow{r} s_1'}{(s_1 \circledast s_2) \xrightarrow{r} (s_2 \circledast s_1')}\ [\textsc{AstStepAns}]$$

**Fig. 5.** Rules for "⊛" evaluation

$$\frac{s_1 \xrightarrow{\circ}_c \Diamond}{(s_1 \oplus s_2) \xrightarrow{\circ} \Diamond}\ [\textsc{SumStopC}] \qquad \frac{s_1 \xrightarrow{r}_c \Diamond}{(s_1 \oplus s_2) \xrightarrow{r} \Diamond}\ [\textsc{SumStopAnsC}]$$

$$\frac{s_1 \xrightarrow{\circ}_c s_1'}{(s_1 \oplus s_2) \xrightarrow{\circ} s_1'}\ [\textsc{SumStepC}] \qquad \frac{s_1 \xrightarrow{r}_c s_1'}{(s_1 \oplus s_2) \xrightarrow{r} s_1'}\ [\textsc{SumStepAnsC}]$$

$$\frac{s_1 \xrightarrow{\circ}_c \Diamond}{(s_1 \circledast s_2) \xrightarrow{\circ}_c \Diamond}\ [\textsc{AstStopC}] \qquad \frac{s_1 \xrightarrow{r}_c \Diamond}{(s_1 \circledast s_2) \xrightarrow{r}_c \Diamond}\ [\textsc{AstStopAnsC}]$$

$$\frac{s_1 \xrightarrow{\circ}_c s_1'}{(s_1 \circledast s_2) \xrightarrow{\circ}_c s_1'}\ [\textsc{AstStepC}] \qquad \frac{s_1 \xrightarrow{r}_c s_1'}{(s_1 \circledast s_2) \xrightarrow{r}_c s_1'}\ [\textsc{AstStepAnsC}]$$

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

$$\frac{s \xrightarrow{\circ}_c s'}{(s \otimes g) \xrightarrow{\circ}_c (s' \otimes g)}\ [\textsc{ProdStepC}] \qquad \frac{s \xrightarrow{(\sigma,n)}_c s'}{(s \otimes g) \xrightarrow{\circ}_c (\langle g, \sigma, n\rangle \circledast (s' \otimes g))}\ [\textsc{ProdStepAnsC}]$$

**Fig. 6.** Cut signal propagation rules

## C SLD Resolution with Cut

The semantics for SLD resolution with cut is built upon our usual semantics for SLD resolution described in section 7.2 using a few additional constructs and rules for them.

We introduce one additional type of goal — "cut". In denotational semantics, we interpret cut as success (thus, denotationally we treat all cuts as green).

Operationally, we modify SLD semantics in such a way that a cut cuts all other branches of all enclosing nodes, marked with "⊕", up to the moment when the evaluation of the disjunct, containing the cut, was started. It is easy to see that this node will always be the nearest "⊕", derived from the disjunction. Unfortunately, in the tree other "⊕" nodes can appear due to the evaluation of "⊗" nodes, thus we need a way to distinguish these two sorts of "⊕". We introduce a separate kind of state "⊛" for the sums of goals created during the evaluation of "⊗" nodes.

Therefore, in the semantics the rule [ProdStepAns] is replaced with the following.

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

The rules for "$\circledast$" evaluation mirror those for "$\oplus$" (see Fig. 5).

To perform the cutting of the branches, we introduce a separate kind of transitions to propagate the signal for cutting (denoted by $\xrightarrow{l}_c$). The signal is risen when a "cut" construct is encountered.

$$\langle !, \sigma, n\rangle \xrightarrow{(\sigma,n)}_c \Diamond\ [\textsc{Cut}]$$

When the signal is being propagated through "$\oplus$" and "$\circledast$" nodes, their right branches are cut out, and for "$\circledast$" the propagation continues; in the case of "$\otimes$" nodes the signal is simply propagated (see Fig. 6).