Certified Semantics for Relational Programming

Dmitry Rozplokhas
Higher School of Economics and JetBrains Research, Russia
darozplokhas@edu.hse.ru

Andrey Vyatkin
Saint Petersburg State University, Russia
dewshick@gmail.com

Dmitry Boulytchev
Saint Petersburg State University and JetBrains Research, Russia
dboulytchev@math.spbu.ru

Abstract
We present a formal study of semantics for relational programming language miniKanren. First, we formulate denotational semantics which corresponds to the minimal Herbrand model for definite logic programs. Second, we present operational semantics which models the distinctive feature of miniKanren implementation — interleaving, — 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.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Denotational semantics; Theory of computation → Operational semantics

Keywords and phrases Relational programming, denotational semantics, operational semantics, certified programming

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. Unlike Prolog, which relies on SLD-resolution, most miniKanren implementations use a monadic interleaving search, which is known to be complete. miniKanren is designed as a shallow DSL which may help to equip the host language with logical reasoning features. This design choice was proven to be applicable in practice, and there are more than 100 implementations for almost 50 languages.

The introductory book on miniKanren describes the language by means of an evolving set of examples. In the series of follow-up papers 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 languages’ semantics and therefore is not stable under the host language replacement. In addition, the justification of important properties of relational programs (for example, refutational completeness) becomes cumbersome.

There were some previous attempts to define a formal semantics for miniKanren. In formal definitions for denotational and non-deterministic operational semantics were given

A detailed Prolog-to-miniKanren comparison can be found here: http://minikanren.org/minikanren-and-prolog.html
and the soundness result was proven; the development was mechanized in HOL. [22] presented a variant of nondeterministic operational semantics, and [27] used another variant of finite-set semantics. None of the previous approaches were capable of reflecting the distinctive property of miniKanren’s search — interleaving [16], thus deviating from the conventional understanding of the language.

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 [21]; then we describe operational semantics with interleaving in terms of a labeled transition system. Finally, we prove the soundness and completeness of the operational semantics w.r.t the denotational one. We support our development with a formal specification using the Coq [3] proof assistant, 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.

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 the 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. Section 6 presents some applications of the previous development, including the certified semantics for SLD resolution with cut. The second to the last section surveys related works; 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 in Figure 1. First, we fix a set of constructors $C$ with known arities and consider a set of terms $\mathcal{T}_X$ with constructors as functional symbols and variables from $X$. We parameterize this set with an alphabet of variables since in the semantic description we will need two kinds of variables. The first kind, syntactic variables,
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. For the sake of brevity we abbreviate immediately nested “fresh” constructs into the one, writing “fresh \( x \ y \ldots \ g \)” instead of “fresh \( x \) . fresh \( y \ldots \) g”. The final syntactic category is a specification \( S \). 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 given in 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 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 “\( g_1 \land g_2 \)” applies \( g_1 \) to the current state and then applies \( g_2 \) to each element of the result, concatenating the streams.
- Disjunction “\( g_1 \lor g_2 \)” 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 “\( R_{ki}^k(t_1, \ldots, t_{ki}) \)” finds a definition for the relational symbol \( R_{ki}^k = \lambda x_1 \ldots x_{ki} \cdot g_i \), 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 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) \[16\]; the evaluation of some goals can be additionally deferred (via so-called “inverse-\(\eta\)-delay”) \[12\]; instead of streams the implementation can be based on “ferns” \[7\] 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:

\[
\text{append}^{o} = \lambda \ x \ y \ xy . \\
((x \equiv \text{Nil}) \land (xy \equiv y)) \lor \\
(fresh \ h \ t \ ty . \\
(x \equiv \text{Cons} (h, t)) \land \\
(xy \equiv \text{Cons} (h, ty)) \land \\
(append^{o} \ t \ y \ ty) \\
) ;
\]

\[
\text{revers}^{o} = \lambda \ x \ y . \\
((x \equiv \text{Nil}) \land (y \equiv \text{Nil})) \lor \\
(fresh \ h \ t \ t' . \\
(x \equiv \text{Cons} (h, t)) \land \\
(append^{o} \ t' (\text{Cons} (h, \text{Nil}) \ y) \land \\
(\text{revers}^{o} \ t \ t') \\
)
\]

\[
\text{revers}^{o} x \ x
\]

Here we defined two relational symbols — “\text{append}^{o}” and “\text{revers}^{o}”, — and specified a top-level goal “\text{revers}^{o} x \ x”. The symbol “\text{append}^{o}” defines a relational concatenation of lists — it takes three arguments and performs a case analysis on the first one. If the first argument is an empty list (“\text{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 “\text{append}^{o}” and additional variable “\(ty\)”, which represents the concatenation of “\(t\)” and “\(y\)”. Finally, we unify “\text{Cons} (h, ty)” with “\(xy\)” to form a final constraint. Similarly, “\text{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:

\[
\alpha \mapsto \text{Nil} \\
\alpha \mapsto \text{Cons} (\beta_0, \text{Nil}) \\
\alpha \mapsto \text{Cons} (\beta_0, \text{Cons} (\beta_0, \text{Nil})) \\
\alpha \mapsto \text{Cons} (\beta_0, \text{Cons} (\beta_1, \text{Cons} (\beta_0, \text{Nil})) \\
\ldots
\]

where “\(\alpha\)” — a semantic variable, corresponding to “\(x\)”, “\(\beta_i\)” — free semantics variables.

The syntax described 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:

\[\text{We respect here a conventional tradition for miniKanren programming to superscript all relational names with }^{o}.\]
Inductive term : Set :=
| Var : name → term
| Cst : name → term
| Con : name → term → term → term.

Here “name” is a type for all named entities (variables, constructors and relations), for which we simply take natural numbers:

Definition name : Set := nat.

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 always represent a sequence of terms as a list using constructors $\text{Nil}^0$ and $\text{Cons}^2$.

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 : (name → goal) → goal
| Invoke : name → term → goal.

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 is more convenient to define a set of free variables as a proposition:

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

Note that in our formalization we use higher-order abstract syntax for variable binding [22], 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 some requirements on the syntax representation which are trivially fulfilled in the first-order case.

First, we need a requirement that the definitions of relations do not contain unbound variables:

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 second requirement is that all bindings have to be “consistent”, i.e. if we instanti-
ate a higher-order “fresh” construct for different 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).
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 with only non-free variables
is sufficient:

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 can now set an arbitrary environment (a map from a relational symbol to a definition
of relation with described constraints) 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.
Variable Prog : env.
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 [21]. 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 [6]) which we plan to do in future.

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 [29]. 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:

\[
x \equiv \text{Cons} \ (y, \ z)
\]

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

\[
\alpha \mapsto \text{Cons} \ (\beta, \ \gamma)
\]

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:
\{(\text{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

\[ f : \mathcal{A} \mapsto \mathcal{D} \]

from semantic variables to ground terms. We call these functions \textit{representing functions}. Thus, we may reformulate the same relation as

\[ \{(f(\alpha), f(\beta), f(\gamma)) \mid f \in [\alpha \equiv \text{Cons} (\beta, \gamma)]\} \]

where we use conventional semantic brackets “[•]” 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

\[ f : \mathcal{A} \rightarrow \mathcal{D} \]

we introduce its homomorphic extension

\[ \overline{f} : \mathcal{T}_\mathcal{A} \rightarrow \mathcal{D} \]

which maps terms to terms:

\[ \overline{f}(\alpha) = f(\alpha) \]

\[ \overline{f}(C^{k_i}_{i}(t_1, \ldots, t_{k_i})) = C^{k_i}_{i}(\overline{f}(t_1), \ldots, \overline{f}(t_{k_i})) \]

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 \(f\) for which \(\overline{f}(t_1) = \overline{f}(t_2)\).

We remind the conventional notions of pointwise modification of a function

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

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

For a representing function \(f : \mathcal{A} \rightarrow \mathcal{D}\) and a semantic variable \(\alpha\) we define the following \textit{generalization} operation:
\[ f \uparrow \alpha = \{ f[\alpha \leftarrow d] \mid d \in D \} \]

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 \( D \). We extend the generalization operation for sets of representing functions \( \mathcal{F} \subseteq A \rightarrow D \):

\[ \mathcal{F} \uparrow \alpha = \bigcup_{f \in \mathcal{F}} (f \uparrow \alpha) \]

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 \( \mathcal{F}(t_1) = \mathcal{F}(t_2) \) 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. Now for the case of “\( \text{fresh} \ x \ . \ g \)”.

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 “\( \text{fresh} \ x \ . \ g \)”

The final case is a relational invocation, in which we unfold the definition of the corresponding relational symbol and substitute its formal parameters with actual ones.

Here is an example of denotational semantics of a goal:
The proof is by induction on evidence. The only non-trivial case is when the goal is

Lemma 2

\[ \exists g, f : \text{in\_denotational\_sem\_goal} \quad g \vdash f \implies f \in [g] \]

We formulate the following important completeness condition for the semantics of a goal \( g \):

Lemma 2 (Completeness condition). For any goal \( g \) and two representing functions \( f \) and \( f' \), such that \( f|_{\text{FV}(g)} = f'|_{\text{FV}(g)} \)

\[ f \in [g] \implies f' \in [g] \]

Proof. The proof is by induction on evidence. The only non-trivial case is when the goal is a "fresh" construct, there we need to change the value of the given function in one point to apply the inductive hypothesis.

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.
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). From now on we assume the set of semantic variables to be linearly ordered ($A = \{\alpha_1, \alpha_2, \ldots \}$).

We introduce the notion of substitution

$$\sigma : A \rightarrow T$$

as a (partial) mapping from semantic variables to terms over the set of semantic variables. We denote $\Sigma$ the set of all substitutions, $\text{Dom}(\sigma)$ — the domain for a substitution $\sigma$, $\text{Vran}(\sigma) = \bigcup_{\alpha \in \text{Dom}(\sigma)} \text{FV}(\sigma(\alpha))$ — 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, $\sigma$ — 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

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

where $\diamond$ symbolizes the end of the evaluation, and the following well-formedness condition:

Definition 3. Well-formedness condition for extended states:

- $\diamond$ is well-formed;
- $\langle g, \sigma, n \rangle$ is well-formed iff $\text{FV}(g) \cup \text{Dom}(\sigma) \cup \text{Vran}(\sigma) \subseteq \{\alpha_1, \ldots, \alpha_n\}$;
- $s_1 \oplus s_2$ 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 $s$ $\text{FV}(g) \subseteq \{\alpha_1, \ldots, \alpha_n\}$.

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:

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

The label "$\diamond$" 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

---

D. Rozplokhas, A. Vyatkin and D. Boulytchev 11
\[ \langle t_1 \equiv t_2, \sigma, n \rangle \xrightarrow{\text{UnifyFail}} \emptyset, \not\exists \text{mgu} (t_1\sigma, t_2\sigma) \]

\[ \langle t_1 \equiv t_2, \sigma, n \rangle \xrightarrow{\text{UnifySuccess}} \emptyset \]

\[ \langle g_1 \lor g_2, \sigma, n \rangle \xrightarrow{\text{Disj}} \langle g_1, \sigma, n \rangle \oplus \langle g_2, \sigma, n \rangle \]

\[ \langle g_1 \land g_2, \sigma, n \rangle \xrightarrow{\text{Conj}} \langle g_1, \sigma, n \rangle \otimes g_2 \]

\[ \langle \text{fresh } x, g, \sigma, n \rangle \xrightarrow{\text{Fresh}} \langle g[\alpha_{n+1}/x], \sigma, n + 1 \rangle \]

\[ \langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{Invoke}} \langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \]

\[ \begin{array}{c}
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{DisjStop}} \emptyset \\
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{DisjStopAns}} \emptyset \\
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{ConjStop}} \emptyset \\
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{ConjStopAns}} \emptyset \\
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{DisjStep}} \emptyset \\
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{DisjStepAns}} \emptyset \\
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{ConjStep}} \emptyset \\
\langle \mathcal{R}^k_i (t_1, \ldots, t_k), \sigma, n \rangle \xrightarrow{\text{ConjStepAns}} \emptyset \\
\end{array} \]

\[ \mathcal{R}^k_i = \lambda x_1 \ldots x_k . g \]

**Figure 5** Operational semantics of interleaving search
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 in Figure 5. 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 answer equal to the most general unifier.

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 “⊕”) for evaluating in the same context as the parent state, for conjunction — schedules the left goal and postpones the right one (using “⊗”).

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.

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 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 (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 that 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”

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.

To formalize the operational part in Coq we first need to define all preliminary notions from unification theory [2] which our semantics uses.

In particular, we need to implement the notion of the most general unifier (MGU). As it is well-known [23] 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

---

3 There still can be differences in observable behavior of concrete goals under different sound and complete search strategies: a goal can be refutationally complete under one strategy and non-complete under another.
termination check failure. There is no such obstacle when we define MGU as a proposition:

\[
\text{Inductive mgu :} \quad \text{term} \rightarrow \text{term} \rightarrow \text{option subst} \rightarrow \text{Set} := ...
\]

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_result_exists:** \(\forall t_1 t_2, \{r \& \text{mgu } t_1 t_2 r\}\).

**Definition unifier (s : subst) (t1 t2 : term) : Prop :=**
\[\text{apply_subst } s \ t_1 = \text{apply_subst } s \ t_2.\]

**Lemma mgu_unifies:**
\(\forall t_1 t_2 s, \text{mgu } t_1 t_2 (\text{Some } s) \rightarrow \text{unifier } s \ t_1 t_2.\)

**Definition more_general (m s : subst) : Prop :=**
\[\exists (s' : \text{subst}), \quad \forall (t : \text{term}), \quad \text{apply_subst } s \ t = \text{apply_subst } s' (\text{apply_subst } m \ t).\]

**Lemma mgu_most_general:**
\(\forall (t_1 t_2 : \text{term}) \quad (m : \text{subst}), \quad \text{mgu } t_1 t_2 (\text{Some } m) \rightarrow \forall (s : \text{subst}), \quad \text{unifier } s \ t_1 t_2 \rightarrow \text{more_general } m \ s.\)

**Lemma mgu_non_unifiable:**
\(\forall (t_1 t_2 : \text{term}), \quad \text{mgu } t_1 t_2 \text{ None } \rightarrow \forall s, \neg (\text{unifier } s \ t_1 t_2).\)

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.

After this preliminary work, the described transition relation can be encoded naturally as an inductively defined proposition (here “state” stands for a non-extended state and “state’” — for an extended one):

\[
\text{Inductive eval_step :} \quad \text{state'} \rightarrow \text{label} \rightarrow \text{state} \rightarrow \text{Set} := ...
\]

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

**Lemma eval_step_exists:**
\(\forall (st' : \text{state'}), \quad \{1 : \text{label} \& \{st : \text{state} \& \text{eval_step } st' 1 \ st}\}.\)

**Lemma eval_step_unique:**
\(\forall (st' : \text{state'}) \quad (l_1 l_2 : \text{label}) \quad (st_1 st_2 : \text{state'}), \quad \text{eval_step } st' l_1 st_1 \rightarrow \text{eval_step } st' l_2 st_2 \rightarrow l_1 = l_2 \land st_1 = st_2.\)
To work with (possibly) infinite sequences we use the standard approach in CoQ — coinductively defined streams:

\[
\begin{align*}
\textbf{Context} & \{ A : \text{Set} \}. \\
\textbf{CoInductive} & \quad \text{stream} : \text{Set} := \\
& | \text{Nil} : \text{stream} \\
& | \text{Cons} : A \rightarrow \text{stream} \rightarrow \text{stream}.
\end{align*}
\]

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

\[
\begin{align*}
\textbf{Inductive} & \quad \text{in_stream} : A \rightarrow \text{stream} \rightarrow \text{Prop} := \\
& | \text{inHead} : \forall x t, \text{in_stream} x (\text{Cons} x t) \\
& | \text{inTail} : \forall x h t, \text{in_stream} x t \rightarrow \text{in_stream} x (\text{Cons} h t).
\end{align*}
\]

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:

\[
\begin{align*}
\textbf{Definition} & \quad \text{trace} : \text{Set} := @\text{stream label}. \\
\textbf{CoInductive} & \quad \text{op_sem} : \text{state} \rightarrow \text{trace} \rightarrow \text{Set} := \\
& | \text{osStop} : \text{op_sem} \text{ Stop} \text{ Nil} \\
& | \text{osState} : \forall \text{ st' l st t}, \text{eval_step st' l st} \rightarrow \\
& \quad \text{op_sem st t} \rightarrow \\
& \quad \text{op_sem (State st') (Cons l t)}.
\end{align*}
\]

\[
\begin{align*}
\textbf{Lemma} & \quad \text{op_sem_exists} (\text{st} : \text{state}) : \\
& \{ t : \text{trace} \& \text{op_sem st t} \}.
\end{align*}
\]

\[
\begin{align*}
\textbf{Lemma} & \quad \text{op_sem_unique} : \\
& \forall st t1 t2, \\
& \quad \text{op_sem st t1} \rightarrow \\
& \quad \text{op_sem st t2} \rightarrow \\
& \quad \text{equal_streams t1 t2}.
\end{align*}
\]

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 [8].

One thing we can prove using operational semantics is the \textit{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:

\[
\begin{align*}
\textbf{CoInductive} & \quad \text{interleave} : \\
& \quad \text{stream} \rightarrow \text{stream} \rightarrow \text{stream} \rightarrow \text{Prop} := \\
& | \text{interNil} : \forall s s', \\
& \quad \text{equal_streams s s'} \rightarrow \text{interleave Nil s s'} \\
& | \text{interCons} : \forall h t s rs, \\
& \quad \text{interleave s t rs} \rightarrow \\
& \quad \text{interleave (Cons h t) s (Cons h rs)}.
\end{align*}
\]
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.

Lemma 4. For any streams s1, s2, s satisfying interleave s1 s2 s, the elements occurring in s are exactly the elements occurring in s1 or s2.

Lemma 5. For any streams s1, s2, s satisfying interleave s1 s2 s, s is finite iff s1 is finite and s2 is finite.

One immediate corollary of these facts is the “commutativity” of disjunction under the interleaving search: the fact that swapping two disjuncts (at the top level) does not change the termination of the goal evaluation.

5 Equivalence of Semantics

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 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 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 which preserves all variable dependencies in the substitution. So we can define a set of representing functions corresponding to a substitution as follows:

\[
[\sigma] = \{ f \circ \sigma \mid f : A \rightarrow D \}
\]

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

\[
[s]_{op} = \bigcup_{(\sigma, n) \in T_r} [\sigma]
\]

This allows us to state theorems relating two semantics.
\[
\begin{align*}
\emptyset & = \emptyset \\
\llbracket (g, \sigma, n) \rrbracket & = \llbracket g \rrbracket \cap \llbracket \sigma \rrbracket \\
\llbracket s_1 \mathbin{\oplus} s_2 \rrbracket & = \llbracket s_1 \rrbracket \cup \llbracket s_2 \rrbracket \\
\llbracket s \mathbin{\otimes} g \rrbracket & = \llbracket s \rrbracket \cap \llbracket g \rrbracket
\end{align*}
\]

\textbf{Figure 6} Denotational semantics of states

\begin{enumerate}
\item \textbf{Theorem 6} (Operational semantics soundness). If indices of all free variables in a goal \( g \) are limited by some number \( n \), then

\[
\llbracket (g, \epsilon, n) \rrbracket \text{_{op}} \subset \llbracket g \rrbracket.
\]

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 shown on Figure 6. The generalized version of the theorem uses it:

\item \textbf{Lemma 7} (Generalized soundness). For any well-formed extended state \( s \)

\[
\llbracket s \rrbracket \text{_{op}} \subset \llbracket s \rrbracket.
\]

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 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.

\item \textbf{Lemma 8} (Soundness of the answer). For any well-formed extended state \( s \) and any transition \( s \xrightarrow{(\sigma, n)} s' \),

\[
\llbracket \sigma \rrbracket \subset \llbracket s \rrbracket.
\]

\textbf{Proof}. The proof is by induction on the transition relation. The only non-trivial case here is the rule UnifySuccess (the only rule where the answer is created, not passed through). Here we need to show that any representing function that is an extension of \( \text{mgu} (t_1\sigma, t_2\sigma) \circ \sigma \) unifies terms \( t_1 \) and \( t_2 \) and is an extension of \( \sigma \). These rather trivial (as long as we have proven the properties of mgu) statements require some technical work with the properties of representing functions.

\item \textbf{Lemma 9} (Soundness of the next state). For any well-formed extended state \( s \) and any transition \( s \xrightarrow{(\sigma, n)} s' \),

\[
\llbracket s' \rrbracket \subset \llbracket s \rrbracket.
\]
**Proof.** The proof is by straightforward induction on the transition relation. In cases where the initial state is a partially-evaluated conjunction we need the previous lemma.

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 completeness condition, while the operational semantics may also contain dependencies between semantic variables allocated in “fresh”. 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 provides the result as substitutions only for queried variables, which are allocated in the beginning.

**Theorem 10 (Operational semantics completeness).** If indices of all free variables in a goal \( g \) are limited by some number \( n \), then

\[
\{ f \mid \{ \alpha_1, \ldots, \alpha_n \} \mid f \in \llbracket g \rrbracket \} \subset \{ f \mid \{ \alpha_1, \ldots, \alpha_n \} \mid f \in \llbracket \langle g, \epsilon, n \rangle \rrbracket_{op} \}.
\]

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 — a bounded denotational semantics:

\[
\llbracket \bullet \rrbracket : \mathcal{G} \rightarrow 2^{A \rightarrow D}
\]

Instead of always unfolding the definition of a relation for invocation goal, it does so only the given number of times. So for a given set of relational definitions \( \{ R_{i}^{k_{i}} = \lambda x_{i}^{1} \ldots x_{i}^{k_{i}} \cdot g_{i} \} \) the definition of bounded denotational semantics is exactly the same as for the conventional denotational semantics, except that for the invocation case we have

\[
\llbracket R_{i}^{k_{i}} (t_{1}, \ldots, t_{k_{i}}) \rrbracket_{l}^{l+1} = \llbracket g_{i}[t_{1}/x_{i}^{1}, \ldots, t_{k_{i}}/x_{i}^{k_{i}}] \rrbracket_{l}
\]

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

\[
\llbracket g \rrbracket_{0} = \emptyset
\]

The bounded denotational semantics is an approximation of the conventional denotational semantics; it is clear that any answer in the conventional denotational semantics will also be in the bounded denotational semantics for some level.

**Lemma 11.** \( [g] \subset \cup_{l} [g]_{l} \)

Formally it can be proven using the definition of the least fixed point from the 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 12 (Generalized completeness).** For any set of relational definitions, for any level \( l \), for any well-formed state \( \langle g, \sigma, n \rangle \),

\[
\{ f \mid \{ \alpha_1, \ldots, \alpha_n \} \mid f \in \llbracket g \rrbracket_{l} \cap [\sigma] \} \subset \{ f \mid \{ \alpha_1, \ldots, \alpha_n \} \mid f \in \llbracket \langle g, \sigma, n \rangle \rrbracket_{op} \}.
\]
Proof. The proof is by the induction on the level $l$. The induction step is proven by structural induction on the goal $g$.

- If the goal is $t_1 \equiv t_2$ 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 $\text{mgu}(t_1\sigma, t_2\sigma) \circ \sigma$. This again 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.

- If the goal is $g_1 \lor g_2$ we use lemma 4 that guarantees that any answer in the trace for $s_1$ or in the trace for $s_2$ will also occur in trace for $s_1 \oplus s_2$.

- If the goal is $g_1 \land g_2$ we use the similar observation about partially-evaluated conjunctions (for any answer $(\sigma, n)$ in the trace starting from $s$, any answer in the trace starting from $(g, \sigma, n)$ will also occur in the trace for $s \otimes g$). We also need the completeness condition and the fact that the number of allocated variables only increases during the execution.

- The case when the goal is fresh $x$. $g'$ is actually very non-trivial because given representing function from the denotational semantics may come from a substitution of fresh-bound variable with any non-free variable, not necessarily the next one allocated. To overcome this difficulty we need the restriction on allocated variables in theorem statement; we also need the “monstrous” lemma 1 about switching fresh variable and have to carefully change the value of a given function in a few points to be able to use the induction hypothesis appropriately.

- If the goal is $R(t_1, \ldots, t_k)$ we simply use the induction hypothesis about the previous level.

\[\square\]

The proofs of both theorems are certified in CoQ; for completeness, we can not just use the induction on proposition $\text{in\_denotational\_sem\_goal}$, as it would be natural to expect, because the inductive principle it provides is not flexible enough. So we need to define the bounded denotational semantics in our formalization and perform induction on the level explicitly.

The final proofs are relatively small, but the large amount of work is hidden in the proofs of auxiliary facts. The total size of the proofs of soundness is 70 LOC and for the proofs of completeness it is roughly 450 LOC (this includes the proofs of observations about representing functions, operational and denotational semantics that are necessary for the final proof and not important on their own, but excluding properties of unifiers, streams and so on).

6 Applications

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

6.1 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 which 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 into 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.

As an example of these transformations application we consider the transformation from PROLOG to conventional miniKANREN and, more interesting, in the opposite direction.

Since in PROLOG a rather limited form of goals (an implicit conjunction of atoms) is used the conversion to miniKANREN is easy:

- the built-in constructors in PROLOG terms (for example, for lists) are converted into miniKANREN representation;
- each atom is converted into a corresponding relational invocation with the same parameters (modulo the conversion of terms);
- the list of atoms in the body of PROLOG clause is converted into explicit conjunction;
- a number of fresh variables (one for each argument) are created
- the arguments are unified with the terms in the corresponding argument position in the head of the corresponding clause;
- different clauses for the same predicate are combined using disjunction.

For example, consider the result of conversion from PROLOG definition for the list appending relation into miniKANREN. Like in miniKANREN, the definition consists of two clauses. The first one is

\[ \text{append} ([], X, X). \]

and the result of conversion is

\[ \text{append} = \lambda x \ y \ z . \]
\[ \text{fresh} X . x == \text{Nil} \land \]
\[ y == X \land \]
\[ z == X \]

The second one is

\[ \text{append} ([H|T], Y, [H|TY]) :- \text{append} (T, Y, TY). \]

which is converted into

\[ \text{append} = \lambda x \ y \ z . \]
\[ \text{fresh} H \ T \ Y \ TY . x == \text{Cons} (H, T) \land \]
\[ y == Y \land \]
\[ z == \text{Cons} (H, TY) \land \]
\[ \text{append} (T, Y, TY) \]

The overall result is not literally the same as what we’ve shown in Section 2, but denotationally equivalent.

The conversion in the opposite direction involves the following steps:

- converting between term representation;
moving all “fresh” constructs into the top-level;

- transforming the freshless body into DNF;

- replacing all unifications with calls for a specific predicate “unify/2”, defined as

\[
\text{unify} (X, X).
\]

- splitting the top-level disjunctions into separate clauses with the same head.

The correctness of these, again, can be justified denotationally. For the append relation in Section 2 the result will be as follows:

\[
\begin{align*}
\text{append} (X, Y, Z) & : = \text{unify} (X, []) , \text{unify} (Z, Y) . \\
\text{append} (X, Y, Z) & : = \\
& \quad \text{unify} (X, [H|T]) , \\
& \quad \text{unify} (Z, [H|TY]) , \\
& \quad \text{append} (T, Y, TY).
\end{align*}
\]

These transformations show that we can, for example, interpret Prolog specifications in interleaving semantics; moreover, we can, using the certified framework we developed, describe conventional Prolog search strategies.

6.2 SLD Semantics

The conventional for 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:

\[
\begin{align*}
\frac{s_1 \rightarrow c \Diamond}{(s_1 \oplus s_2) \rightarrow (s_1' \oplus s_2)} & \quad \text{[DisjStep]} \\
\frac{(s_1 \oplus s_2) \rightarrow (s_1' \oplus s_2)}{s_1 \rightarrow s_1'} & \quad \text{[DisjStepAns]} \\
\end{align*}
\]

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).
6.3 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 verbose). 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”.

First, we add “cut” as a new sort of goals:

\[
\text{Inductive goal : Set := } \ldots \mid \text{Cut : } \text{goal.}
\]

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 denote the new sort of nodes as “⊛”, and modify the definition of states.

In the semantics the rule \([\text{CONJSTEPANS}]\) is replaced with

\[
\begin{array}{c}
(s \otimes g) \xrightarrow{\sigma} (s') \quad \text{[CONJSTEPANS]}
\end{array}
\]

The rules for “⊛” evaluation mirror those for “⊕”, so we omit them.

We need a separate kind of transitions to propagate the signal for cutting the branches \(\xrightarrow{\sigma C/\rho_c} (s \otimes g) \xrightarrow{\rho_c} \).

The signal itself is risen when a “cut” construct is encountered:

\[
\langle ! \rangle, \sigma, n \xrightarrow{\sigma C/\rho_c} \quad \text{[Cut]}
\]

When the signal is being propagated through “⊕” and “⊛” nodes, their right branches are cut out, and for “⊗” the propagation continues (see Fig. 7). In the case of “⊗” nodes the signal is simply propagated; we omit the rules since they mirror the regular ones.

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 of extracted reference interpreter using “litmus tests”.

Consider the following specification:

1. \(a(0)\).
2. \(a(1)\).
3. \(b(2, 0)\).
Strictly speaking, integer literals can not be used in the language we defined; however we can replace them by Peano-encoded integers with the same result. The query we are interested in is “c(W, X, Y, Z)”.

| W   | X   | Y   | Z   |
|-----|-----|-----|-----|
| 4   | 0   | 0   | 0   |
| 0   | 2   | 0   | 0   |
| 0   | 0   | 2   | 0   |
| 0   | 0   | 0   | 0   |
| 0   | 0   | 0   | 1   |
| 0   | 1   | 0   | 0   |
| 0   | 1   | 1   | 0   |
| 1   | 2   | 0   | 0   |
| 1   | 2   | 0   | 1   |
| 1   | 0   | 0   | 0   |
| 1   | 0   | 1   | 0   |
| 1   | 0   | 1   | 1   |
| 5   | 0   | 0   | 0   |

These answers are exactly those (including the order) delivered by SWI-Prolog.

### 6.4 Reference Interpreters

Using 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 behaviour of specifications in unclear, complex or corner cases. Our experience has shown that these interpreters demonstrate the expected behavior in all cases.

### 7 Related Works

The study of formal semantics for logic programming languages, in the first place Prolog, is a well-established research domain. Early works 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 the equivalence of a few differently defined operational semantics for pure Prolog is proven, and in a denotational semantics for Prolog with cut is presented; both works provide Coq-mechanised proofs. It is interesting that the former one also advocates the use of higher-order abstract syntax. We are not aware of any prior works 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 [24, 17, 26], where the descriptions of unification algorithms are given in LCF, Alf, Coq and Coq respectively. As a well-founded order lexicographically ordered tuples, containing the information about the number of different free variables and the sizes of the arguments, is used. We implemented a similar approach, but we separated 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 order. Alternatively, in [23] a structurally recursive definition of unification algorithm is given; 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 earlier work [10], where it was employed to describe lambda calculus. The inconsistency phenomenon of HOAS representation, mentioned in Section 2, is called there “exotic terms” and is handled using a dedicated inductive predicate “Valid_v”. The predicate has a non-trivial implementation based on subtle observations on bindings behavior. 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 predicate “consistent_binding” can be considered as a limited version of “Valid_v” for a smaller language.

The study of formal semantics for miniKanren is also not a completely novel venture. In [22] a non-deterministic small-step semantics is described, and in [27] a big-step semantics for a finite number of answers is given; neither uses proof mechanization and in both works the interleaving is not addressed.

The most important property of interleaving search — completeness — was postulated in the introductory paper [16], and is delivered by all major implementations. In [14] a formal proof of completeness is presented; however, the completeness is understood as a preservation of all answers during the interleaving of answer streams, i.e. in a more narrow sense than in our case since no relation to denotational semantics is established; additionally, no proof mechanization is used.

The work of [20] 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 nondeterministic (similarly to [22]), 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.

8 Conclusion and Future Work

In this paper we presented a formal semantics for core miniKanren and proved some of its basic properties (including interleaving search completeness), 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 Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. cKanren: miniKanren with constraints. In Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming, October 2011.
2 Franz Baader and Wayne Snyder. Handbook of automated reasoning. chapter Unification Theory. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.
3 Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. URL: https://doi.org/10.1007/978-3-662-07964-5
4 Ana 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.
5 William E. Byrd. Relational Programming in miniKanren: Techniques, Applications, and Implementations. PhD thesis, Indiana University, September 2009.
6 William E. Byrd and Daniel P. Friedman. okanren: A fresh name in nominal logic programming. In Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming, pages 79–90, 2007.
7 William E. Byrd, Daniel P. Friedman, Ramana Kumar, and Joseph 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.
8 Adam Chlipala. Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant. MIT Press, 2013. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types
9 Saumya K. Debray and Prateek 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.
10 Joëlle Despeyroux, Amy P. Felty, and André 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. URL: https://doi.org/10.1007/BFb0014049 doi:10.1007/BFb0014049
11 Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. The reasoned schemer. MIT Press, 2005.
12 Jason Hemann and Daniel P. Friedman. jkanren: A minimal functional core for relational programming. In Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming, 2013.
13 Jason Hemann and Daniel 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
14 Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew 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. URL: https://doi.org/10.1145/2989225.2989230
doi:10.1145/2989225.2989230

15 Neil D. Jones and Alan 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.

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

17 Sunil Kothari and James 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. URL: https://doi.org/10.1007/978-3-7045-0921-7_3

18 Jael Kriener and Andy King. Semantics for prolog with cut - revisited. In Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, Japan 4-6, 2014, Proceedings, pages 270–284, 2014. URL: https://doi.org/10.1007/978-3-319-07151-0_17

19 Jael Kriener, Andy King, and Sandrine 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. URL: https://doi.org/10.1145/2505879.2505886 doi:10.1145/2505879.2505886

20 Ramana Kumar. Mechanising aspects of minikanren in hol. Bachelor Thesis, The Australian National University, 2010.

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

22 Petr Lozov, Andrei Vyatkin, and Dmitry 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. URL: https://doi.org/10.1007/978-3-319-89717-8_3 doi:10.1007/978-3-319-89719-6_3

23 Conor McBride. First-order unification by structural recursion. J. Funct. Program., 13(6):1061–1075, 2003. URL: https://doi.org/10.1017/S0956796803004957

24 Lawrence C. Paulson. Verifying the unification algorithm in LCF. Sci. Comput. Program., 5(2):143–169, 1985. URL: https://doi.org/10.1016/0167-6423(85)90009-7

25 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. pages 199–208, 1988. URL: https://doi.org/10.1145/53990.54010 doi:10.1145/53990.54010

26 Rodrigo Geraldo Ribeiro and Carlos 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. URL: https://doi.org/10.1007/978-3-319-29473-5_8 doi:10.1007/978-3-319-29473-5_8

27 Dmitri Rozplokhas and Dmitri 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 181-18:13, 2018. URL: https://doi.org/10.1145/3236950.3236958 doi:10.1145/3236950.3236958
28 Cameron Swords and Daniel P. Friedman. rkanren: Guided search in minikanren. In Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming, 2013.

29 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, 5, 06 1955. doi:10.2140/pjm.1955.5.285

30 Philip 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. URL: https://doi.org/10.1007/3-540-59451-5_2 doi:10.1007/3-540-59451-5_2