Visualizing miniKanren Search with a Fine-Grained Small-Step Semantics

BRYSEN PFINGSTEN, Seton Hall University, USA
JASON HEMANN, Seton Hall University, USA

We present a deterministic small-step operational semantics for miniKanren that explicitly represents the evolving search tree during execution. This semantics models interleaving and goal scheduling at fine granularity, allowing each evaluation step—goal activation, suspension, resumption, and success—to be visualized precisely. Building on this model, we implement an interactive visualizer that renders the search tree as it develops and lets users step through execution. The tool acts as a pedagogical notional machine for reasoning about miniKanren's fair search behavior, helping users understand surprising answer orders and operational effects. Our semantics and tool are validated through property-based testing and illustrated with several examples.

ACM Reference Format:
Brysen Pfingsten and Jason Hemann. 2025. Visualizing miniKanren Search with a Fine-Grained Small-Step Semantics. 1, 1 (October 2025), 21 pages.

1 Introduction

The miniKanren family of logic programming languages uses a fair interleaving depth-first search strategy [31] to find solutions. The execution of a logic program involves a complex, non-deterministic search process, making it difficult for a programmer to predict the order in which a given program will search or how performant that search will be. Small differences in goal ordering or structuring can dramatically alter the order in which answers appear or the time it takes to compute them. Implementers too may find it difficult to anticipate the consequences of even slightly altering where their implementation places delays. For example, consider the miniKanren query in Figure 1.

```
(defrel (same x y) (== x y))

(run* (q)
  (conde
    [(conde
       [(same q 'turtle)]
       [(same q 'cat)]
       [(== q 'dog)])]
    [(same q 'fish)]))
```

```
same(X,Y) :- X = Y.

?- ( ( same(Q,turtle)
     ; same(Q,cat)
     ; Q = dog
     )
   ; same(Q,fish)
   )
```

Fig. 1. Definitions and uses of the same relation, written in both Friedman et al. [24]'s concrete miniKanren syntax as well as in Prolog.

Intuitively, a programmer might expect the answer `turtle` to appear first, because it appears earliest in the source code. The Prolog implementation produces `Q = turtle`, `Q = cat`, `Q = dog`, and `Q =`

Authors' Contact Information: Brysen Pfingsten, pfingsbr@shu.edu, Seton Hall University, USA; Jason Hemann, jason.hemann@shu.edu, Seton Hall University, USA.

ACM XXXX-XXXX/2025/10-ART
https://doi.org/

`fish` in that order. Running this query in a miniKanren like those described by Friedman et al. [24] or Hemann and Friedman [26], however, produces `'(fish turtle dog cat)`. In practice, miniKanren can easily produce answers in an unintuitive order due to its fair scheduler and interleaving strategy.

Without a clear understanding of the operational semantics, the reasons behind such behavior remain obscure. Declarative debugging techniques [9, 46] offer limited help here, as the issue is not logical correctness—all branches eventually succeed—but rather the operational details of how the search is conducted.

In this paper, we address precisely this challenge by introducing a **deterministic small-step operational semantics** for miniKanren that *explicitly encodes the evolving search tree*. We model the nondeterministic miniKanren computation as deterministic transformations of a search tree. Unlike existing formal semantics for logic programming languages [17, 33], our approach models miniKanren's fair interleaving, and does so at a *fine-grained, intra-goal* level. More specifically, our semantics exposes each distinct evaluation step within the execution of individual miniKanren goals, including goal initiation, suspension, resumption, and final success or failure. Previous work on miniKanren semantics either models interleaving at a higher, coarser granularity—stepping directly from one completed relational call to another without exposing the crucial internal decision points—or uses a simpler approximation of typical implementations' search behavior [41, 43]. Our approach, by contrast, transforms the search tree at each intra-goal evaluation step, (clarifying when, how, and why each goal is executed, paused, or resumed), and models the more parsimonious interleave behavior of existing implementations.

We use these semantics as the basis for an interactive **search-tree expression stepper**, inspired by expression steppers from the programming languages education literature [12, 13, 14, 36]. This tool functions as a pedagogical *notional machine* [18] for miniKanren, explicitly visualizing the evolving search tree at every evaluation step. Students and new users can interactively step forward and backward through the search process, inspecting the current state, suspended goals, and chosen branches. We hope that in doing so, they build an accurate mental model of how miniKanren conducts its fair interleaving search and better understand how logic programs execute. While primarily designed as an educational visualizer, our tool may also provide experienced miniKanren programmers with limited diagnostic capabilities. The prototype is available online (minikanrenredex-prod.shu.edu) for the community to test and use.

Using PLT Redex [22, 32], we built a **mechanized implementation** of our semantics. Our tool is built around this mechanization, which we use as an executable reference interpreter. This mechanization is also a platform for automated testing of key properties like determinism and preservation of well-formedness.

To demonstrate our semantics and visualization approach concretely, we present detailed stepping traces of miniKanren queries, including the unintuitive ordering scenario above. Our semantics framework can accommodate alternate search strategies; we validate this by also implementing a Prolog-style depth-first search by simply swapping out the reduction rules. We believe this approach provides a valuable pedagogical foundation for teaching miniKanren's operational behavior, as well as a sound semantic basis for future exploration of logic programming execution strategies.

The remainder of the paper proceeds as follows. Section 2 reviews miniKanren's syntax, gives a high-level intuitive comparison of miniKanren's operational semantics to that of Prolog, and reprises interleaving search strategies. Section 3 defines our formal small-step semantics explicitly, explaining the search-tree representation and detailed reduction rules. Section 4 describes our interactive visualizer tool's design goals, implementation, its integration with the Redex semantics, and our design choices to improve usability. In Section 5, we illustrate the semantics and visualizer in action with concrete miniKanren queries, showing how stepping through execution can illuminate otherwise confusing search behaviors. Finally, Section 6 situates our contributions within existing work on logic programming visualization and semantics, and Section 7 discusses future directions and potential extensions of this approach.

2 Background

In this section, we discuss some background and ideas from CS education research. We revisit traditional Prolog traces and the Byrd box model. We also re-introduce miniKanren and its semantics informally, by way of a contrast with Prolog and traditional logic programming.

2.1 Visual steppers and notional machines

Students' difficulty in understanding logic program execution—what is sometimes referred to as the "Prolog story" problem—has been well documented in computing education research [38, 45, 48]. Learners often struggle to form mental models of a language's non-deterministic control flow, especially backtracking, goal scheduling, and the order of answer production. Over the years, researchers and educators have explored a wide variety of tracers and visualization tools to help make the execution process more accessible.

A *notional machine* [18] is a simplified conceptual model used in education to help learners reason about program execution. Similarly, a *visual stepper* is a pedagogical tool allowing users to step through program evaluation visually, forming correct mental models by observing the evolving state. Tools like the PLT Redex stepper have proven valuable pedagogically precisely because they expose each small step of computation explicitly and clearly.

2.2 Traditional Prolog execution models and debugging

Early Prolog debuggers were built around the box model, introduced by Byrd [8] as a way to help users understand Prolog's control flow. In the four-port Byrd model, each predicate call is conceptualized as a "box" that goes through ports: typically Call, Exit (success), Fail, and Redo. This model underlies most Prolog tracers allowing the programmer to single-step through each call and backtracking event. The Byrd box model's simplicity made it ubiquitous, but it can produce very verbose traces and requires the user to mentally reconstruct the program's logical structure from low-level steps. To provide higher-level views of execution, researchers turned to AND/OR tree representations. An AND/OR tree depicts the search space of a query as a tree of goals: an AND-node's children (placed side-by-side) are all the sub-goals of a conjunction, and an OR-node's children are alternative goal expansions. Each path down an AND/OR tree corresponds to a potential solution, and backtracking corresponds to exploring alternate branches (OR-nodes) of the tree. Tree models like this make backtracking points and the overall search structure explicit, which is harder to see in the linear Byrd port trace. However, they can become large and cluttered for complex programs, and relating them back to code (and features like the cut) is non-trivial.

2.3 miniKanren's simplified execution model

The miniKanren language is simpler—both more verbose and less expressive—than more traditional logic or constraint-logic programming languages. It differs from traditional Prolog-style logic languages in several key ways that together simplify the language's execution model, making it particularly amenable to a fine-grained, explicit semantics as we develop in Section 3. The language assumes a fixed set of relational definitions with no dynamic `assert` or `retract`. Calls to undefined relations or relation-arity mismatches are treated as errors, and can in some cases be caught statically by the host language. Unlike Prolog, miniKanren has no implicit head unification for relations. The programmer writes each predicate directly in a form of program completion [11, 37]. Relations are invoked via explicit parameter substitution (akin to β-substitution in functional languages), and all unification is done explicitly via goals.

The miniKanren term language is first-order, meaning relations themselves are not in the domain of terms and there are no higher-order relations. The programmer explicitly introduces new logic variables. Lexical variables (local to a relation) and logical variables (introduced implicitly by `fresh`) have clearly disjoint scopes. Associating a lexical variable with its corresponding logic variable can also be modeled by β-substitution.

The core language is small enough that the example in Figure 1 exercises most of the miniKanren language forms. Negation is generally disallowed, and in most dialects limited to a handful of negated primitive constraints on terms [25, 35]. The miniKanren programming community emphasizes pure, cut-free declarative programming, which aligns with many modern Prolog best practices [49]. Even the miniKanren arithmetic suite [30] is implemented as library routines over bit-vectors that behave reasonably in all modes. Host-language macro extension facilities are used to provide less cumbersome surface forms that de-sugar to the core. User-level host-language macros scratch much the same itch as meta-programming in Prolog.

3 Language and Semantics

In this section we present our model of miniKanren execution, which we define by a reduction semantics. We assume familiarity with standard reduction semantics terminology and notation, as described in, e.g., the PLT Redex textbook [22]. Readers primarily interested in the visualization aspects or high-level intuition may skip ahead without significant difficulty.

Unlike typical miniKanren implementations that use implicit streams of answers, our semantics encodes the entire search tree and its relation environment in a single expression. The main technical contribution described in this section is treating a nondeterministic search process as a deterministic state transition system over a tree. We get a deterministic single-step transition that still models nondeterministic search, by threading the whole search tree through states. We first describe the grammar of programs, relations, and search trees, then explain the reduction rules and how they realize the standard miniKanren search strategy.

3.1 Language

Figure 2 shows the grammar of our core language. Our grammar consists of a core microKanren-like language extended with explicit constructs to represent the state of the search: notably, a search tree (S) data structure is a component of the program. A program is a triple `prg` Σ S where Σ is the *relation environment* and S is a *search tree*. As usual each relation definition in Σ is given as $r(L) \leftrightarrow G$ with r a relation name, L a list of syntactic variables (parameters), and G a goal that constitutes the relation's body.

We assume the existence of several pairwise-disjoint infinite sets: syntactic variables, relation names, logic variables (written l_n which represents the n^{th} logic variable introduced in a given state), and constants. The full term language consists of the closure of a binary cons constructor $:$ over the union of the syntactic variables, logic variables, and constants. Our miniKanren language provides a primitive goal $\top$ that always succeeds, and five primitive constructors for term equations, relation calls, binary conjunction, binary disjunction, and variable introduction.

A pair of a goal G and a *state* σ is itself a search tree. A state σ is a pair (θ, i) where θ is a *substitution* mapping logic variables to terms and i is a numeric index. To initialize the computation the programmer's `(run (L) G . . . . )` query is transformed into an existentially quantified fresh goal $\exists$ `(L) G ...`, and injected into a pair with the *initial state*. The initial state consists of an empty substitution and 0.

Every leaf node of a search tree is either of the form $G\ \sigma$ or EMPTY, which represents an empty search tree node. Complex search trees are built from simpler search trees using search tree combinators. Binary constructors $S \longrightarrow S$ and $S \longleftarrow S$ represent pointed disjunctions, where the arrow depicts the

$$
\begin{array}{rl@{\qquad}rl}
\text{Programs} & \ni p \coloneqq \mathtt{prg}\ \Sigma\ S & \mathbb{N} & \ni i \\
\text{Environments} & \ni \Sigma \coloneqq r(L) \leftrightarrow G \ldots & \text{Constants} & \ni c \\
\text{Parameter Lists} & \ni L \coloneqq x \ldots & \text{Variables} & \ni x, y \\
\text{Search Trees} & \ni S \coloneqq \mathtt{empty} \mid G\ \sigma \mid S \rightarrow S \mid S \leftarrow S & \text{Relation Names} & \ni r \\
 & \quad \mid S + S \mid S \times G & \text{Logic Variables} & \ni l_i \\
 & \quad \mid \mathtt{go}\ S & \text{Terms} & t \coloneqq c \mid x \mid l_i \mid t : t \\
 & \quad \mid \mathtt{delay}\ S & \text{States} & \sigma \coloneqq (\theta, i) \\
\text{Goals} & \ni G \coloneqq \top \mid t \equiv t \mid r(t \ldots) & \text{Substitutions} & \theta \coloneqq (l_i \mapsto t) \ldots \\
 & \quad \mid G \vee G \mid G \wedge G \mid \exists\ (L)\ G & &
\end{array}
$$

Fig. 2. Core grammar of our language

search ordering of its children. S + S is an undirected disjunction, and only appears in an answer stream. S × G represents a conjunction of a search tree with a goal. Computation proceeds first down the left child; the goal on the right-hand side does not have a state because its state is determined by the search down the left-hand side. Special constructors `delay S` and `go S` mark the "delay points" for fair interleaving, points where the engine can suspend/resume goals.

We specify a set of evaluation contexts in Figure 3 that identify the next reducible subexpression within a program. The program context E_P focuses on the search tree inside of a program; all

$$
\begin{aligned}
E_P &\coloneqq \mathtt{prg}\ \Sigma\ \square \\
E_A &\coloneqq \square \mid (\top\ \sigma) + E_A \\
E_S &\coloneqq \square \mid E_S \leftarrow S \mid S \rightarrow E_S \mid E_S \times G \\
E &\coloneqq E_P[E_A[E_S\ \square]]
\end{aligned}
$$

Fig. 3. Evaluation contexts

reductions occur within the search tree. An answer stream context E_A brings the focus to the search tree just below the sequence of finished answers. A search context E_S brings the focus to the node within the search tree that is to be reduced. Intuitively, this context descends into the search tree following the directive of disjunctions and the tree component of conjunctions. Finally, we define E, the composition of the previous three contexts to locate within a program the next search tree node to reduce.

3.2 Semantics

Figure 4 presents the small-step reduction rules for our μKanren inspired language. We model the miniKanren search behavior described in Hemann and Friedman [26]. In our semantics the program state contains an explicit representation of the entire search tree as it is being explored. The benefit is that the reduction rules can manipulate this tree structure to simulate goal interleaving (for example, by taking a suspended goal from one branch and moving it to an active position in another branch). Our semantics explicitly represents individual steps within the execution of each miniKanren goal, such as initiating the evaluation of a goal, suspending it, resuming it, and ultimately completing it (success or failure). We refer to these individual evaluation points as intra-goal evaluation steps. Most of the reduction rules in our semantics utilize the E context with the exception of those which match on the relation environment (Proceed) and those which must occur just below the answer stream (InvokeDelay, PromoteLeft/Right).

$$\textsc{DistrDisj}\quad E[(G_1 \vee G_2)\,\sigma] \longrightarrow E[(G_1\,\sigma) \leftarrow (G_2\,\sigma)]$$

$$\textsc{DistrConj}\quad E[(G_1 \wedge G_2)\,\sigma] \longrightarrow E[(G_1\,\sigma) \times G_2]$$

$$\textsc{LeftAnsConj}\quad E[((\top\,\sigma) \leftarrow S) \times G] \longrightarrow E[((\top\,\sigma) \times G) \leftarrow (S \times G)]$$

$$\textsc{RightAnsConj}\quad E[(S \rightarrow (\top\,\sigma)) \times G] \longrightarrow E[(S \times G) \rightarrow ((\top\,\sigma) \times G)]$$

$$\textsc{AssocRightLeft}\quad E[S_1 \rightarrow ((\top\,\sigma) \leftarrow S_2)] \longrightarrow E[(\top\,\sigma) \leftarrow (S_1 \rightarrow S_2)]$$

$$\textsc{AssocRightRight}\quad E[S_2 \rightarrow (S_1 \rightarrow (\top\,\sigma))] \longrightarrow E[(S_2 \rightarrow S_1) \rightarrow (\top\,\sigma)]$$

$$\textsc{AssocLeftLeft}\quad E[((\top\,\sigma) \leftarrow S_1) \leftarrow S_2] \longrightarrow E[(\top\,\sigma) \leftarrow (S_1 \leftarrow S_2)]$$

$$\textsc{AssocLeftRight}\quad E[(S_1 \rightarrow (\top\,\sigma)) \leftarrow S_2] \longrightarrow E[(S_1 \leftarrow S_2) \rightarrow (\top\,\sigma)]$$

$$\textsc{SuccConj}\quad E[(\top\,\sigma) \times G] \longrightarrow E[G\,\sigma]$$

$$\textsc{PruneConj}\quad E[\texttt{empty} \times G] \longrightarrow E[\texttt{empty}]$$

$$\textsc{PruneLeft}\quad E[\texttt{empty} \leftarrow S] \longrightarrow E[S]$$

$$\textsc{PruneRight}\quad E[S \rightarrow \texttt{empty}] \longrightarrow E[S]$$

$$\textsc{SubstFresh}\quad \frac{i' = i + |x\ldots| \qquad G' = G[x\ldots/l_i\ldots]}{E[(\exists\ (x\ldots)\ G)\ (\theta, i)] \longrightarrow E[G'\ (\theta, i')]}$$

$$\textsc{Delay}\quad E[r(t_1\ldots)\,\sigma] \longrightarrow E[\texttt{delay}\ (\texttt{go}\ r(t_1\ldots)\,\sigma)]$$

$$\textsc{Proceed}\quad \frac{G' = G[x\ldots/t\ldots]}{(prg\ (\ldots r(x\ldots) \leftrightarrow G\ldots)E_A[E_S[\texttt{go}\ r(t\ldots)\ \sigma]]) \longrightarrow (prg\ (\ldots r(x\ldots) \leftrightarrow G\ldots)E_A[E_S[G'\,\sigma]])}$$

$$\textsc{UnifySucc}\quad \frac{\theta' = \text{unify}(t_1, t_2, \theta)}{E[((t_1 \equiv t_2)\ (\theta, i))] \longrightarrow E[(\top\ (\theta', i))]}$$

$$\textsc{UnifyFail}\quad \frac{\nexists\ \text{mgu}(t_1, t_2, \theta)}{E[((t_1 \equiv t_2)\ (\theta, i))] \longrightarrow E[\texttt{empty}]}$$

$$\textsc{DelayConj}\quad E[((\texttt{delay}\ S)\ \times\ G)] \longrightarrow E[(\texttt{delay}\ (S\ \times\ G))]$$

$$\textsc{DelayLeft}\quad E[((\texttt{delay}\ S_1) \leftarrow S_2)] \longrightarrow E[(\texttt{delay}\ (S_1 \rightarrow\ S_2))]$$

$$\textsc{DelayRight}\quad E[(S_1 \rightarrow\ (\texttt{delay}\ S_2))] \longrightarrow E[(\texttt{delay}\ (S_1 \leftarrow\ S_2))]$$

$$\textsc{InvokeDelay}\quad E_P[E_A[(\texttt{delay}\ S)]] \longrightarrow E_P[E_A[S]]$$

$$\textsc{PromoteLeft}\quad E_P[E_A[((\top\,\sigma) \leftarrow S)]] \longrightarrow E_P[E_A[((\top\,\sigma) + S)]]$$

$$\textsc{PromoteRight}\quad E_P[E_A[(S \rightarrow (\top\,\sigma))]] \longrightarrow E_P[E_A[((\top\,\sigma) + S)]]$$

Fig. 4. microKanren language semantics

We limit the amount of interleaving by introducing delay points only at relation invocations (Delay). This results in a more biased, less interleaved search than Rozplokhas et al.'s [44] model. Figure 5 shows a trace demonstrating how delay and forcing of delays behaves on a small example.

$$\begin{array}{rl}
 & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\ \ (\exists(p)\ \texttt{same}(p,\texttt{'cat}))\ (\emptyset, 0) \\
\Rightarrow^{\textsc{SubstFresh}} & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\ \ \texttt{same}(0,\texttt{'cat})\ (\emptyset, 1) \\
\Rightarrow^{\textsc{Delay}} & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\ \ \texttt{delay}(\texttt{go}(\texttt{same}(0,\texttt{'cat})))\ (\emptyset, 1) \\
\Rightarrow^{\textsc{InvokeDelay}} & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\ \ \texttt{go}(\texttt{same}(0,\texttt{'cat}))\ (\emptyset, 1) \\
\Rightarrow^{\textsc{Proceed}} & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\ \ 0 \equiv \texttt{'cat}\ (\emptyset, 1) \\
\Rightarrow^{\textsc{UnifySucc}} & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\ \ \top\ ((0 \mapsto \texttt{'cat}), 1)
\end{array}$$

Fig. 5. Worked example trace: evaluating $(\exists(p)\ \texttt{same}(p,\texttt{'cat}))$ under our semantics.

To show how the interleaving and resumption machinery behaves, we now sketch a trace for a query with a disjunction. This example will show how fair interleaving is implemented by step-wise promotion and resume. Delayed subtrees are explicitly wrapped with a `delay` tag and bubbled upward where they interleave disjunctions (DelayLeft/Right) and bypass conjunctions (DelayConj). Once the delay tag reaches the tail of the answer stream, it is released (InvokeDelay) and evaluation continues in the freshly interleaved tree. Figure 6 shows key steps in evaluating a disjunctive query with two `same` calls, illustrating how our semantics models promotion and fair interleaving of branches.

$$\begin{aligned}
& \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\;\; (\texttt{same}(0,'\texttt{cat}) \vee \texttt{same}(0,'\texttt{dog}))\;(\emptyset,1) \\
\Rightarrow^{\textsc{DistrDisj}}\; & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\;\; (\texttt{same}(0,'\texttt{cat})(\emptyset,1) \leftarrow \texttt{same}(0,'\texttt{dog})(\emptyset,1) \\
\Rightarrow^{\textsc{Delay + Go (Left branch)}}\; & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\;\; \texttt{delay}(\texttt{go}(\texttt{same}(0,'\texttt{cat}))) \leftarrow \texttt{same}(0,'\texttt{dog})\;(\emptyset,1) \\
\Rightarrow^{\textsc{Proceed + UnifySucc}}\; & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\;\; \top\;((0 \mapsto '\texttt{cat}),1) \leftarrow \texttt{same}(0,'\texttt{dog})\;(\emptyset,1) \\
\Rightarrow^{\textsc{PromoteRight}}\; & \texttt{prg}\,(\texttt{same}(x,y) \leftrightarrow x \equiv y)\;\; (\top\;((0 \mapsto '\texttt{cat}),1) + \texttt{same}(0,'\texttt{dog}))\;(\emptyset,1)
\end{aligned}$$

Fig. 6. Trace for `(conde ((same q 'cat)) ((same q 'dog)))`, showing interleaving promotion.

3.3 Novel design choices

Our small-step semantics were designed with visual debugging in mind, which influenced many decisions regarding the language.

Explicit Answer Stream. Successful goals are not immediately removed from the search unlike that of Rozplokhas et al. [44]. Instead, a success is represented as $(\top\,\sigma)$ and remains as part of the structure until it is lifted to the top-level answer stream (PromoteLeft/Right). We model the answer stream as a sequence of these answer nodes accumulated with the + operator. This allows the user to inspect each answer as it is produced and to see it in context. It also preserves a record of answers which can be useful for understanding the progress of the search. While success is explicitly modeled, failure (represented as `empty`) is not. Upon unification failing (UnifyFail), that subtree is promptly pruned from the tree (PruneConj, PruneLeftRight) mimicking the silent refutation of miniKanren.

Pointed Disjunction (Railway Model). We represent disjunctions in a symmetric, bidirectional way to preserve the relative orderings of disjuncts upon interleaving (DelayLeft/Right). We refer to this as the *railway model* as the two branches of a disjunction are like a switch in railway tracks where evaluation can proceed down either side. In the grammar, these are represented as $\leftarrow$ and $\rightarrow$. Traditionally, a singular left-biased disjunction is used and interleaving involves swapping the two disjuncts. Visually and textually, this makes it more difficult to retain information about the search tree and requires the user to recalibrate to its new shape. While changing a $\leftarrow$ to a $\rightarrow$ or vice versa is a subtle syntactic shift, it promotes better structural correspondence between successive reductions—particularly when evaluation contexts are made explicit. As seen in Figure 4, this combinatorially affects the number of certain reduction steps (AssocLeftLeft, etc., PromoteLeft/Right, Left/RightAnsConj, DelayLeft/Right, PruneLeft/Right); however, a correctness preserving yet smaller semantics could be derived by collapsing the bidirectional disjunctions into a single unoriented disjunction operator.

Delayed Relation Expansion. In addition to the `delay` operator, we also use a special go operator that wraps relation invocations. This thunk-like form prevents the immediate expansion of a relations body (Delay) until it is scheduled for evaluation (Proceed). By delaying unnecessary expansion, we avoid exploding the search tree too early, making the focus on the current evaluation less cluttered while maintaining a more intuitive conceptualization of the actual goal being suspended (i.e., a relation call with certain arguments rather than whatever its goal body may be). The way that we minimized the expansion of relcalls until we actually use them (see Section 5 for an example). Further debugging-related extensions, including state and goal-level source mapping and additional state information, are described in Section 4.

3.3.1 Semantics validation: Using PLT Redex, we tested two key properties of our reduction semantics. We defined a well-formedness predicate on our search states (ensuring, for example, that paused goals carry a valid state, variables are fresh where expected, etc.). Using that, we were able to test the following:

- **Determinism:** Starting from any well-formed initial configuration, the reduction relation has at most one possible next step. In other words, the semantics is a deterministic small-step interpreter for miniKanren—there are no ambiguous rule overlaps or nondeterministic choices. We validated this by random generation of states and checking that Redex never found two distinct reductions for the same state.
- **Syntactic Preservation** Using Redex's random testing, we confirmed that every reduction step preserves this invariant—only well-formed states produce well-formed successor states. This gives confidence that the formal rules faithfully maintain the intended structure of the search tree, and do not lead to malformed or inconsistent states as the search proceeds.

While well short of a formal proof, these property tests serve as valuable sanity checks. Together, the Redex validation suggests that our semantics is sound and behaves as expected for a wide range of cases.

3.3.2 Alternative interleave strategies. Our semantics implement interleaved exploration of different branches of the search tree. As long as some delay will occur along any cycle in the call graph, no single disjunct or recursive path monopolizes computation, and no potential solution is starved indefinitely. This guarantees completeness even in the presence of infinite branches.

Various miniKanren implementations differ in where they place delay points. These choices directly impact the search order, and so impact queries' efficiency. A key benefit of reduction semantics is their modularity [23]. In our case, this means we can easily swap out reduction relations to explore different search strategies.

To that end, we implement an alternative semantics that uses Prolog's depth-first search within the same framework. It uses an interleaving depth-first search strategy to enumerate substitutions that satisfy a given goal. In contrast with miniKanren's interleaving search, the depth-first search (DFS) of Prolog is more efficient but incomplete. By interleaving nowhere, we can model the DFS search semantics, which serves as a baseline for comparison and to illustrate the ease of modeling alternative strategies in our model.

4 A Mixed Redex-JS Visualizer

We provide a web-based interface with a custom tree visualization that utilizes the executable semantics in our Redex model. By visualizing the search in tree form, the user can see the entire state of the search and observe the operational search behavior including the branching of goals, where goals are suspended or resumed, how execution proceeds incrementally, how state is propagated and expanded, and more. While visual tools of this kind are not inherently better than textual ones [39], the tree structure of our visualizer is more faithful to the nature of the computation and information can be presented in a more comprehensible manner. We paid special attention to addressing usability issues noted in studies of earlier logic-program visualizers (Section 6). For example, our interface explicitly marks the current point of execution (solving the common problem of "unclear current goal" in static tree diagrams) and clearly displays the variable bindings in the current state (making the construction of output substitutions transparent).

Primarily, our visualizer serves as a pedagogical stepping visualizer—a "notional machine" that helps students precisely understand how miniKanren executes logic programs step by step. While the tool provides insight into the evaluation process and search order, it is not a full-fledged debugger with advanced breakpoint or query capabilities. Nonetheless, experienced users might employ it as a limited diagnostic aid for manually tracing and understanding surprising search behaviors. In this section we describe the visualizer including additional infrastructure beyond the Redex model, augmentations to our language and semantics, as well as components of the UI front-end.

4.1 Architecture

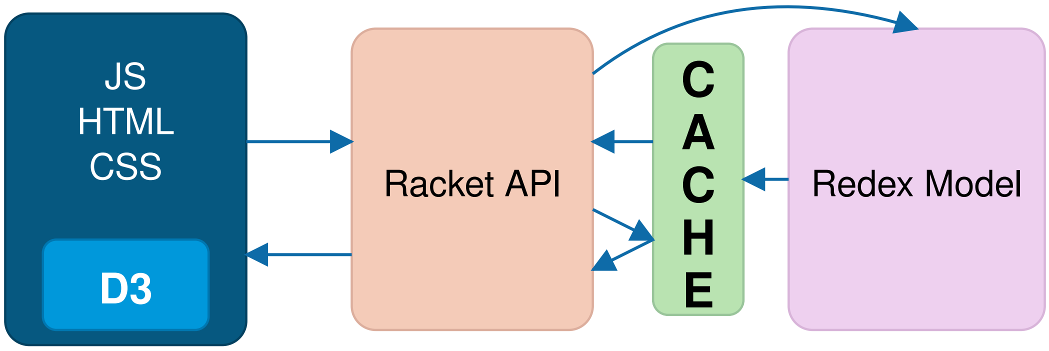


Fig. 7. The architecture of our visualizer.

Figure 7 shows the high-level architecture of visualizer. A User requests such as submission of a program to be visualized, stepping forwards or backwards, resetting, or changing semantics is sent from the frontend to the backend API. Users begin by selecting a set of reductions (currently either μKanren and depth-first). The user then submits a program written in the concrete syntax of Friedman et al. [24] Having a canonical and sugared surface syntax makes this tool accessible for a wide audience without having to learn a new syntax or rewrite existing programs. Additionally, users can explore different semantics over the same exact program base. Upon receiving a new program, a check leveraging `hosted-minikanren` and the `syntax-spec` frontend [3] supplemented by a custom well-formed judgment is performed to detect static errors. This will capture errors such as illegal expressions, parentheses mismatches, relation arity mistakes, and binding errors. If there is such an error, it is reported back to the user and visualization will not proceed until a valid program is entered. Once a valid program is entered, it is transpiled to the language of our model. At this point, goals and the initial state are uniquely tagged to facilitate tracing (see Section 4.2).

Once a program has been loaded, execution proceeds incrementally: each step forward applies a single reduction rule from the model, serializes the updated search tree as JSON, and transmits it to the front end for rendering via the D^3 library [6]. We maintain a list with a zipper [27] as a cache of previous states allowing users to step backwards (without a cache this would not be possible) and forwards through the history. This allows us constant time access for stepping forwards and backwards through the cache as well as insertion at the cost of linear space. If the forward cache is empty and the user requests to step forward, the current program is stepped and cached while the new program gets sent to the front end.

4.2 Language Improvements

The language and semantics described in Section 3 provides a minimal core of interleaving search which we augment with additional information to aid the visualizing experience. This methodology is similar to other examples in the literature such as Bernstein and Stark [4] and Kamburjan et al. [28]. It should be noted that these changes result in an equivalent semantics.

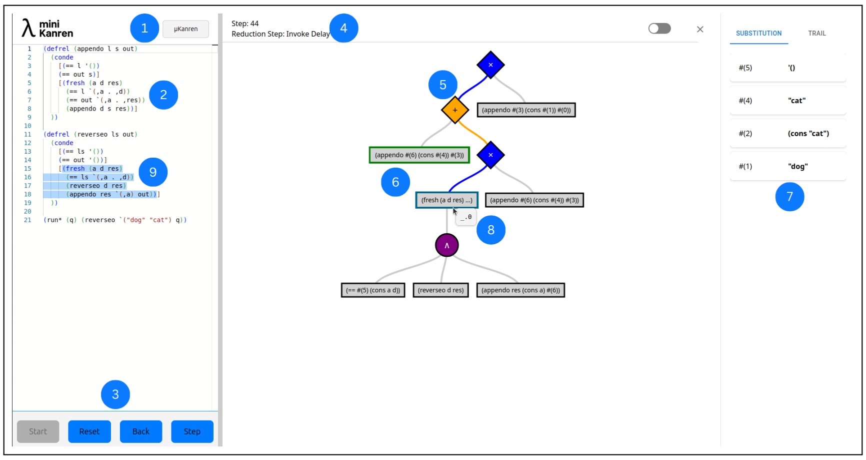
Goal and State Tagging. As mentioned in Section 4.1, goals and states are given UIDs at transpilation time. For goals, this allows for users to bidirectionally trace between the source code they provided and the dynamic tree visualization. This is useful for locating easily locating problematic goals or for understanding where execution is occurring. For states, this enables "subscribing" to a state and following it as it proceeds through execution. Taking unification as an example, the previous non-terminal $t_1 \equiv t_2$ becomes $t_1 \equiv t_2\, c$ where c is the UID. Other goals follow similarly except for $\top$ which remains untagged as it has no mapping to the source code. The state non-terminal now becomes (θ, i, c). The reduction rules that need modified are those dealing with goal-state evaluations and DistrDisj which must also generate a new UID for the state of its right disjunct.

Trail. We once again extend the state terminal with what we refer to as a *trail*. This means a state is now a (θ, i, τ, c) where $\tau = (t_1, t_2, c) \ldots$. A trail is like a substitution except it is always extended by a successful unification and is linearly ordered. This effectively keeps a record of all the atomic computations that have been applied on a given state. As the trail retains the UIDs of the unifications, elements of the trail can also be traced back to the source code. To support this, the UnifySucc rule must append the triple of the walked terms and the UID to the current accumulated trail.

Reification. Although not an extension of the language, we also reify our substitutions before sending them to the visualizer. To do so, we construct a `run*` with the set of initial query variables and a `fresh` for the logic variables created at runtime. Inside the fresh, unifications of the mappings of the substitution are conjuncted. This can result in at most one answer as each substitution represents at most one answer. This is obviously useful for nodes in the answer stream, but we also perform reification on states with more work to be done. This can be useful as the reified representation can reveal information about constraints on the solution such as "this solution must be a pair, although the elements of that pair is not yet known" or "this solution must be 'dog and anything that comes along and contradicts that will cause this branch to fail".

4.3 Visualizer UI

Figure 8 shows the interface for the visualizer. The user begins by selecting a set of reductions (1) and entering their program (2). Once the program is entered, the user can hit the *start* button and begin stepping through execution (3). At every evaluation step, the visualizer presents a rich snapshot of program state. Above the tree is displayed the current step count and the name of the last applied reduction rule (4). The evaluation context is visible by means of tracing the colored edges down to a subtree (5). Here, you can also see the railway model as the disjunction node in orange is right-facing. Additionally, stateful nodes—those which are highlighted (6)—are interactive: clicking on such a node reveals a sidebar with the current substitution and trail (7). Its reified form with respect to the original query variables can be seen by hovering over such a node (8). Finally, goals can be traced bidirectionally from the source code to the tree visualization by simply clicking on them (9). Clicking on a goal in the source code will highlight all occurrences of that goal (if any) in the tree, and clicking on a goal in the tree will highlight the corresponding goal in the source. A full description of the nodes and their meanings is available in Figure 9.



```
(defrel (appendo l s out)
  (conde
    [(== l '())
    (== out s)]
    [(fresh (a d res)
      (== l `(,a . ,d))
      (== out `(,a . ,res))
      (appendo d s res))]
  ))

(defrel (reverseo ls out)
  (conde
    [(== ls '())
    (== out '())]
    [(fresh (a d res)
      (== ls `(,a . ,d))
      (reverseo d res)
      (appendo res `(,a) out))]
  ))

(run* (q) (reverseo `("dog" "cat") q))
```

Step: 44
Reduction Step: Invoke Delay

(appendo #(3) (cons #(1)) #(0))
(appendo #(6) (cons #(4)) #(3))
(fresh (a d res) ...)
(appendo #(6) (cons #(4)) #(3))
(== #(5) (cons a d))
(reverseo d res)
(appendo res (cons a) #(6))

SUBSTITUTION TRAIL

#(5)	'()
#(4)	"cat"
#(2)	(cons "cat")
#(1)	"dog"

Fig. 8. The visualizer.

5 Visualizing Reductions

As discussed in Section 1, understanding why programs produce answers in specific orders is crucial for reasoning about logic programs effectively. In this section, we demonstrate how to leverage our custom visualizer to inspect and understand search behaviors clearly. By stepping through evaluation with both depth-first and interleaving semantics, users can directly observe how different operational semantics influence the order of answers generated by a query. Through illustrative examples, we showcase how this visualization aids users in understanding search scheduling, debugging faulty implementations, and optimizing performance by identifying differences in goal evaluation order.

5.1 Using the visualizer

Understanding Search Behavior. As promised, we will explore why the program in Section 1 produces answers in the order it does.

First, we will explore the execution with the depth-first search semantics which enumerates answers in a more expected manner. Figure 10 shows the execution of the program with arrows indicating the number of intermediate steps that took place. We begin with a top level fresh query with disjunctions in its body. The initial state is distributed over the disjunctions and the evaluation context proceeds down the left side until a relation call is reached. This call, `(sameo #(0) 'turtle)`, is the first to succeed and is brought to the top of the answer stream. Evaluation continues down the left side, distributing the state over another disjunction and encountering `(sameo #(0) 'cat)` which succeeds and is lifted into the answer stream. Finally, the goal `(== #(0) 'dog)` succeeds and is promoted to the answer stream and `(sameo #(0) 'fish)` follows. With this example, the search is straightforward as it is a pure depth-first search with no interleaving. The order of the answers is the same as the order in which the clauses were written.

Figure 11 shows the execution of the same program except with interleaving semantics. The first divergence between these semantics and the depth-first one occurs after four steps: we have substituted our query variable (now `#(0)`), distributed the initial state over two conjunctions, but now the first relation call is delayed. This delay is propagated up, interleaving disjunctions as it bubbles up, until evaluation proceeds down the other side and again delays a relation call. One can already see the railway model and lazy relation expansions at this point. Although interleaving has occurred, the ordering of the disjuncts remains consistent. Additionally, the goals highlighted in green and marked

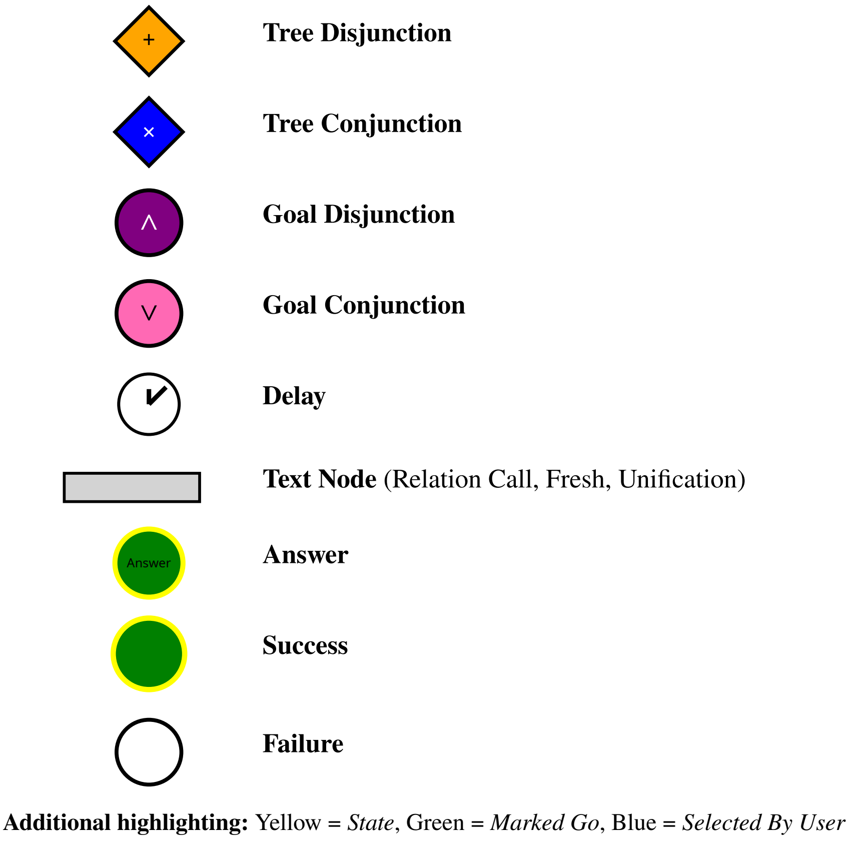

+
Tree Disjunction
×
Tree Conjunction
∧
Goal Disjunction
∨
Goal Conjunction
Delay
Text Node (Relation Call, Fresh, Unification)
Answer
Answer
Success
Failure
Additional highlighting: Yellow = State, Green = Marked Go, Blue = Selected By User

Fig. 9. Node shape legend used in the visualizer.

as ready to proceed by the scheduler have not been expanded. Again, the delay is propagated up and evaluation now proceeds through two disjunctions to yet another relation call. The process repeats and evaluation is now focused on `(sameo #(0) 'fish)` which will succeed and produce the first answer, `'fish`. Evaluation continues down the left side of the tree where `(sameo #(0) 'turtle)` is encountered and produces the second answer, `'turtle`. Next, the right side of the final disjunction, `(== #(0) 'dog)`, immediately succeeds without delay since it is not a relation call and yields the third answer, `'fish`. Finally, the last goal, `(sameo #(0) 'cat)`, succeeds and produces the final answer, `'cat`.

Debugging a Broken Program. As the story goes [41], there was once a novice miniKanren programmer named Nub Let who wrote the following program known by locals as *appendoh the deficient*.

```
(defrel (appendoh l s ls)
  (conde
   ((== '() l) (== s ls))
   ((fresh (a d res)
       (== `(,a . ,d) l)
       (== `(,a . ,res) ls)
```

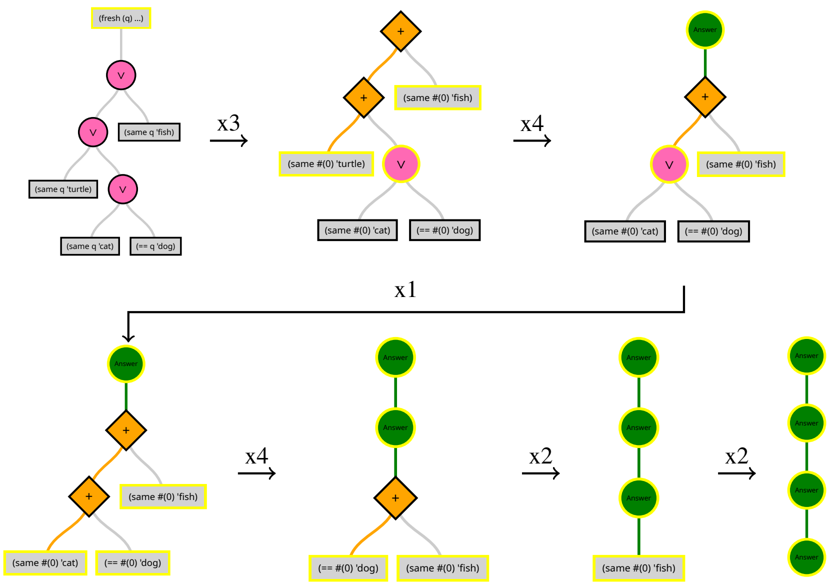
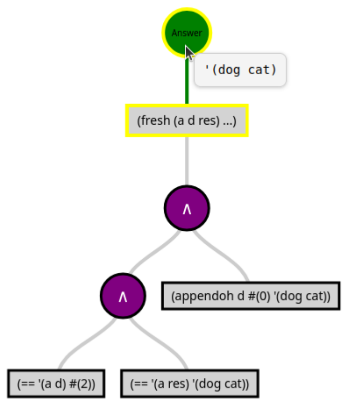


Fig. 10. Evaluation showing DFS behavior.

```
        (appendoh d s ls)))))
```

As a sanity check, Nub Let ran the query `(run* (q) (appendoh '(dog) q '(dog cat)))` but they got out the answer `'((dog cat))`! Clearly, something went wrong, so Nub Let threw the program into our visualizer to find the issue.

Nub Let quickly steps the program until they see the first answer appear in the answer stream. They hover over the node and see the same answer they got before.



Nub Let decides to inspect the trail to see if there are any clues there. They see that the first argument they provided, `'(dog)`, was unified with `(cons #(1) #(2))` and similarly the third argument they provided, `'(dog cat)`, was unified with `(cons #(1) #(3))`. Nub Let maps these to the source program and sees they occurred in the first and second goals within the `fresh`, respectively. Nub Let also observes that #(2) and the empty list were unified along with #(0) and `'(dog cat)` within the

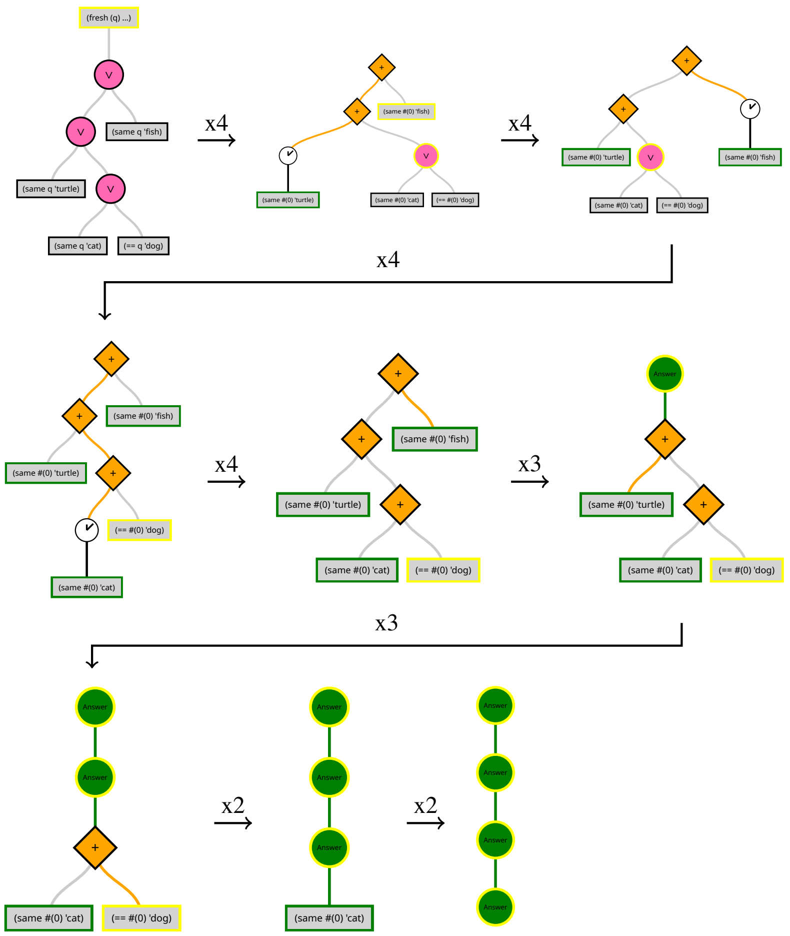
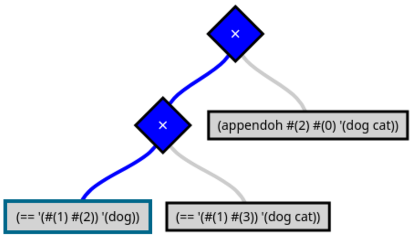


Fig. 11. Evaluation showing interleaving behavior.

first clause of the conde. Nub Let is not experienced enough to be suspicious that #(3) corresponding to the fresh variable `res` introduced in the `fresh` does not appear in this last unification.

Nub Let decides to subscribe to this state by clicking on the answer node and steps backwards looking for the bug. After stepping back a few steps, they observe that they are decomposing the first and third arguments into their head and tails and then recursively calling the appendoh relation.



Here, Nub Let notices that while the first argument is being shrunk in the recursion, the third element is the same as it was in the query! They step past the two unifications and observe the substitution.

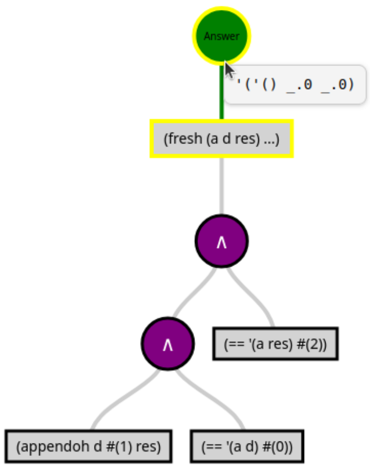
They see #(3)—corresponding to the fresh variable `res`—is mapped to `'cat` and that this is what they should be recurring with. They click on the appendoh node, locate it in their source program, and finally change the `ls` to be `res`.

Optimizing Performance. Nub Let has a cousin named Newb Let who is a seasoned logician and who knows that conjunction is commutative. Like his cousin, Newb Let is adventuring into logic programming and decides to write his own implementation of appendo. They figure it should not matter where the recursive call is placed as the order of conjuncts should not matter and the interleaving search will ensure completeness.

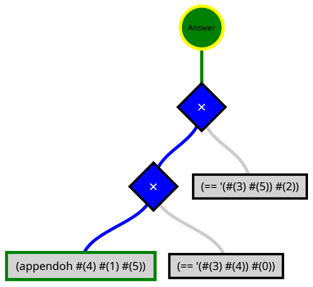
```
(defrel (appendoh l s ls)
  (conde
   ((== '() l) (== s ls))
   ((fresh (a d res)
        (appendoh d s res)
        (== `(,a . ,d) l)
        (== `(,a . ,res) ls)))))
```

However, after benchmarking their implementation, Newb Let finds theirs to be deficient as well! Besides not terminating in some modes, Newb Let's implementation is dramatically slower than the canonical implementation in other modes. They decide to query their relation with all arguments fresh to diagnose the performance bottleneck.

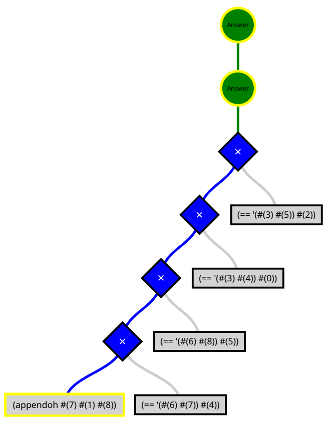
At first, Newb Let observes an answer appears very quickly originating from the first clause.



They are satisfied with this very general answer and proceed with stepping. After more stepping, the first recursive call to appendoh is reached.



While this recursive call did yield another answer, Newb Let notices the goals within the conjunction are still suspended.



As Newb Let continues stepping they get more answers, but it is conjunctions all the way down! Newb Let realizes that each recursive call results in two more conjunctions which a state has to pass through in order to become an answer. Interleaving is of no help here, since conjunctions do not permit interleaving. These conjunctions cannot be pruned either as the recursive call to appendoh continually generates more work to be done. Newb Let learns a lesson and now places his recursive calls at the bottom of his conjunctions.

6 Related Work

We restrict our focus here to fine-grained visualizations of a logic program's execution—those that attempt to model or expose the dynamic computation process as a search tree, rather than simply annotate a linear trace. Most early systems did not support real-time visualization, and could only visualize pre-executed computations. Brna et al. [7, § 2] and Ducassé and Noyé [19, § 7.1] survey the landscape of such early tools. In many cases, such a tool is designed as part of a more featureful debugger. We address here specifically only a few of the most relevant visualizers.

Among the earliest graphical systems in this space is the Transparent Prolog Machine (TPM) [20, 21], which introduced AORTA diagrams (AND/OR trees [34] augmented with status annotations) to depict both the structure of the Prolog search and the control state at each node. TPM's model combined a static representation of the program structure with a depiction of the current state of the ongoing search process. That general approach, depicting computation as movement through an AND/OR-like tree, has informed many later visualizers. TPM itself was pedagogically oriented and aimed to clarify backtracking and cut behavior. Later variants implemented a “live mode” that allowed a user to trace their computation as it ran. In their approach, each individual image in the visualization was partly-dynamic depiction of computation process over time. These images displayed both a search tree, and the sequential process through which it was produced. They indicate that a goal previously failed and was being tried again upon backtracking with so called “ghost nodes”—depicted as shadows behind the image of the goal in the tree. Their model also followed Prolog’s standard depth-first search and did not permit modification of the search. Tamir et al.’s PROVIS [47] targets pure Prolog. Their system provides a split display of program source and a tree. In their case the tree is an SLD resolution tree, representing the computation thus far. Buttons allowed users to interact directly with the GUI. Their visualizer decorates arrows in the SLD tree with the substitution at that point in the computation. Different substitutions can decorate the arrows along a single path from the root, reflecting the accumulation of knowledge as conjuncts are executed in order. For performance reasons, they omit the occurs check. Their tool allowed users to pause execution at disjunctions and explicitly select which branch to explore (so called “debugging on the tree”), giving users limited control over the search path and enabled them to avoid known-diverging branches. Cameron et al. [10] introduce a layered AND/OR representation that captures call structure and backtracking in a dynamic, stepwise fashion. They represent the current state of the search using an AND/OR style tree, representing prior failed attempts in the third dimension. A numbered label in the tree indicates the number of previous backtracked states and clicking it makes the prior tree available. Their tool, ViMer, is a visual debugger for Mercury, incrementally builds a visual representation of execution as the program runs.

Unlike the previous approaches, and ours, Rajan [40] and Adachi et al. [1] do not emphasize the search tree per se. Rajan describes a single stepper for Prolog called APT that, during dynamic execution, correlates the goal being executed with the relevant lines in the source program. APT has a status bar to tell the user what operation the trace is currently performing. Like ours, their visualization is derived from the user’s source code, beginning with the query and evolving with the search process. In the trace representation of the executing query, they use direct substitution both at a clause level (to illustrate a head match) and at the level of individual variables (showing assignments that take place during unification). They therefore single-step at a sub-unification level. Adachi et al.’s Logichart lays out Prolog clauses and calls in a chart-style diagram aligned with source code structure. The execution is animated by color-coded highlighting of elements as they are invoked or completed. This addressed concerns about the abstractness of SLD-trees and aimed to help learners relate behavior to program text.

Our work comes out of a slightly different tradition and with a distinct emphasis. Rather than treat the search tree as a byproduct of evaluation to be visualized post-hoc or recorded for replay, our evolving search tree is explicitly represented in the program state. The tree itself is constructed incrementally. Each tree image corresponds exactly to an intermediate computation state, and every step in the visualizer corresponds 1-1 with a reduction step. This makes it possible to reflect scheduling and interleaving decisions directly in the visualization, not just as an animated trace, but as state-transforming computation.

6.1 Operational semantic models for miniKanren

Several prior efforts have proposed formal or mechanized models of miniKanren execution. Rosenblatt et al. [41] describe a first-order version of miniKanren and used it to implement a stepper that supports user-directed exploration. Their aim is to produce a tooling-friendly representation of miniKanren’s goals and search state. Their approach supports interactive stepping and human-guided search, but does not formalize the search order or represent the interleaving scheduler explicitly. The stepping operates at the goal level, as interpreted by an underlying engine, rather than being grounded in a formal reduction semantics.

Rozplokhas and Boulytchev [42] present a certified operational semantics for miniKanren, formalized and proved correct in Coq. Their semantics ensures fair search by construction, but does so via a maximally interleaving strategy that differs in detail from how typical miniKanren systems behave. That work is oriented toward proof of correctness and completeness, not toward visualization or close-up inspection of the search process.

7 Conclusions and Future Work

In this work we provide a small-step semantics that models miniKanren's interleaving strategy driven by a focus on implementation behavior and pedagogical inspection. In practice, both novice and experienced miniKanren programmers encounter surprising behaviors or performance pathologies due to subtle differences in how goals are written or ordered [41]. The search tree and interleaving control are made explicit in the semantics. Each visual step reflects a concrete transition in that semantics, exposing interleaving decisions and waiting goals. We treat the semantics itself as a pedagogical artifact, a notional machine that can be visualized and stepped through. The resulting tool supports tracing interleaving behavior and reasoning about surprising answer orders, such as those shown in Section 1. We imagine continuing this work in several possible directions.

One possible focus of future work would extending this tool into being a full-fledged debugger. Some of these involve adding new UI features. Being able to "fast forward" or "back up" to the next relation call would let a user approximate Rosenblatt et al.'s goal-level stepping behavior. Leaving invisible marks for the relations that were called, so you can track the trace and order procedure calls which led to that state. Our tool currently shows every small step; this yields comprehensive traces but in lengthy executions could be unwieldy—an open question is how to maintain clarity at scale. One approach would be to add a UI toggle to see "just the current goal", "just the current goal and it's waiting conjuncts", or the whole search tree. We could also imagine adding a profiler based around the search tree execution.

One direction we're definitely interested in is user studies to test our hypotheses about this tool's efficacy for students. User studies and AB-testing could help inform what default UI changes and what new options would help students. For instance, do students do better with the railroad model of interleaving or actually flipping the interleaved subtree across the vertical axis. It may be that the tree flipping could make following the transformation more difficult, but at the same time it might clarify the relationship to traditional depth-first search. The Racket algebraic stepper shows both the before and after of a transformation on the same screen: would that be preferable to our one-tree-at-a-time view? Currently we represent logic variables in the UI literally *as* numeric constants. Choices for naming of logic variables that make sense for the reader to look at but do not confuse with names used by the programmer are a tricky issue. Ducassé and Noyé [19, §4.1] describe this as a well-known problem. Still, we think there's room for UI improvement there. Another question involves the cost of more user-friendly surface syntax. A great virtue of miniKanren as an embedded DSL is that programmers can invent their own custom language extensions host-language using host-language extension mechanisms [2]. Those layers of syntax sugar could, however, open a wide gulf between concise surface syntax using some nice special forms (e.g. `matche` [29]), and the nested binary tree notation we present to users.

Another future direction would be to continue modeling the interleave position design space. The miniKanren community has tried out many different varieties of interleave positions. This implementation approach nicely parameterizes the search behavior, and it would be nice to have all the common choices side-by-side in a common framework that facilitates experimentation.

Finally, we are also interested to explore other semantic artifacts using this language model. From our reduction semantics, we should be able to follow the refocusing, transition compression, fusion sequence that Danvy and Nielsen [16] describe, and generate the microKanren big-step abstract machine. Biernacki and Danvy [5] transform a semi-compositional "double-barrel" CPSed interpreter for a small Prolog-like language into a logic engine by defunctionalization. It will be interesting then to both compare our abstract machine with Biernacki and Danvy's, and also to generate both the corresponding microKanren continuation-passing interpreter and natural semantics [15]. These would in turn help extend Wand and Vaillancourt's [50] work relating models of backtracking to the context of backtracking with interleaving.

References

[1] Yoshihiro Adachi, Kensei Tsuchida, Takanori Imaki, and Takeo Yaku. 1999. Logichart - intelligible program diagram for prolog and its processing system. In *Tenth Workshop on Logic Programming Environments, WLPE 1999, in connection with the International Conference on Logic Programming, ICLP 1999, Las Cruces, New Mexico, USA, November 29, 1999*, 276–288. doi:10.1016/S1571-0661(05)80662-0.

[2] Michael Ballantyne, Mitch Gamburg, and Jason Hemann. 2024. Compiled, extensible, multi-language dsls (functional pearl). *Proc. ACM Program. Lang.*, 8, ICFP, Article 238, (Aug. 2024). doi:10.1145/3674627.

[3] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for domain-specific languages. *Proc. ACM Program. Lang.*, 4, OOPSLA, Article 229, (Nov. 2020). doi:10.1145/3428297.

[4] Karen L. Bernstein and Eugene W. Stark. 1995. Operational semantics of a focusing debugger. *Electronic Notes in Theoretical Computer Science*, 1, 13–31. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. doi:10.1016/S1571-0661(04)80002-1.

[5] Dariusz Biernacki and Olivier Danvy. 2003. From interpreter to logic engine by defunctionalization. In *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, 143–159. doi:10.1007/978-3-540-25938-1_13.

[6] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D³ data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17, 12, 2301–2309. doi:10.1109/TVCG.2011.185.

[7] Paul Brna, Mike Brayshaw, Alan Bundy, et al. 1991. An overview of prolog debugging tools. *Instructional Science*, 20, 2/3, 193–214. An overview/classification of tools that provide different degrees of activity in the debugging process. Other possible dimensions of analysis are also outlined. Retrieved May 23, 2025 from http://www.jstor.org/stable/23369896.

[8] Lawrence Byrd. 1980. Understanding the control flow of prolog programs. In *Proc. of the Logic Programing Workshop*.

[9] Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A survey of algorithmic debugging. *ACM Comput. Surv.*, 50, 4, Article 60, (Aug. 2017). doi:10.1145/3106740.

[10] M. Cameron, M. García de la Banda, K. Marriott, and P. Moulder. 2003. Vimer: a visual debugger for mercury. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (PPDP '03). Association for Computing Machinery, Uppsala, Sweden, 56–66. doi:10.1145/888251.888258.

[11] Keith L. Clark. 1977. Negation as failure. In *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977.* 293–322. doi:10.1007/978-1-4684-3384-5_11.

[12] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an algebraic stepper. In *European symposium on programming*. Springer, 320–334.

[13] John Clements and Shriram Krishnamurthi. 2022. Towards a notional machine for runtime stacks and scope: when stacks don't stack up. In *ICER 2022: ACM Conference on International Computing Education Research, Lugano and Virtual Event, Switzerland, August 7 - 11, 2022, Volume 1*, 206–222. doi:10.1145/3501385.3543961.

[14] Ryan Culpepper and Matthias Felleisen. 2007. Debugging macros. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering* (GPCE '07). Association for Computing Machinery, Salzburg, Austria, 135–144. doi:10.1145/1289971.1289994.

[15] Olivier Danvy. 2005. From reduction-based to reduction-free normalization. *Electronic Notes in Theoretical Computer Science*, 124, 2, 79–100.

[16] Olivier Danvy and Lasse R Nielsen. 2004. Refocusing in reduction semantics. *BRICS Report Series*, 11, 26.

[17] Erik P. de Vink. 1990. Comparative semantics for prolog with cut. *Science of Computer Programming*, 13, 2-3, 237–264.

[18] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14, 3, 237–249. doi:10.1016/S0020-7373(81)80056-9.

[19] Mireille Ducassé and Jacques Noyé. 1994. Logic programming environments: dynamic program analysis and debugging. *The Journal of Logic Programming*, 19, 351–384.

[20] Marc Eisenstadt and Mike Brayshaw. 1990. A fine-grained account of prolog execution for teaching and debugging. *Instructional Science*, 19, 4/5, 407–436. Retrieved May 23, 2025 from http://www.jstor.org/stable/23370487.

[21] Marc Eisenstadt and Mike Brayshaw. 1988. The transparent prolog machine (TPM): an execution model and graphical debugger for logic programming. *J. Log. Program.*, 5, 4, 277–342. doi:10.1016/0743-1066(88)90001-5.

[22] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. (1st ed.). MIT Press. ISBN: 0262062755.

[23] Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103, 2, 235–271.

[24] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition*. (2nd ed.). MIT Press. ISBN: 0262535513.

[25] Jason Hemann. 2020. *Constraint microKanren in the CLP Scheme*. Ph.D. Dissertation. Indiana University, Bloomington, (Jan. 2020). Retrieved July 25, 2025 from https://scholarworks.iu.edu/dspace/handle/2022/25183.

[26] Jason Hemann and Daniel P. Friedman. 2023. Nearly macro-free microkanren. In *Trends in Functional Programming*. Stephen Chang, (Ed.) Springer Berlin Heidelberg, Cham, 72–91. doi:10.1007/978-3-031-38938-2_5.

[27] Gérard Huet. 1997. The zipper. *Journal of Functional Programming*, 7, 5, 549–554. doi:10.1017/S0956796897002864.

[28] Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, et al. 2021. Programming and debugging with semantically lifted states. In *The Semantic Web*. Ruben Verborgh, Katja Hose, Heiko Paulheim, et al., (Eds.) Springer Berlin Heidelberg, Cham, 126–142. ISBN: 978-3-030-77385-4.

[29] Andrew W Keep, Michael D Adams, Lindsey Kuper, et al. 2009. A pattern matcher for miniKanren or how to get into trouble with CPS macros. In *Scheme '09 Workshop on Scheme and Functional Programming* (Scheme '09), 37–45. https://digitalcommons.calpoly.edu/csse_fac/83.

[30] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *Proc. International Symposium on Functional and Logic Programming*, 64–80. doi:10.1007/978-3-540-78969-7_7.

[31] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of ICFP '05* number 9. Vol. 40. Association for Computing Machinery, (Oct. 2005), 192–203. doi:10.1145/1086365.1086390.

[32] Casey Klein, John Clements, Christos Dimoulas, et al. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, 285–296. doi:10.1145/2103656.2103691.

[33] Robert Kowalski and Donald Kuehner. 1971. Linear resolution with selection function. *Artificial Intelligence*, 2, 3-4, 227–260.

[34] Robert A. Kowalski. 1979. *Logic for problem solving*. *The computer science library: Artificial intelligence series*. Vol. 7. North-Holland, New York. ISBN: 0444003681.

[35] Jean-Louis Lassez and Ken McAloon. 1989. Independence of negative constraints. In *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'89)*, 19–27. doi:10.1007/3-540-50939-9_122.

[36] Henry Lieberman. 1984. Steps toward better debugging tools for lisp. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, 247–255.

[37] J.W. Lloyd and R.W. Topor. 1984. Making prolog more expressive. *The Journal of Logic Programming*, 1, 3, 225–240. doi:10.1016/0743-1066(84)90011-6.

[38] Robert Hawley, (Ed.) 1987. *What stories should we tell novice prolog programmers? Artificial Intelligence Programming Environments*. John Wiley & Sons, Inc., USA, 119–130. ISBN: 0470209895.

[39] Mukesh J. Patel, Chris Taylor, and Benedict du Boulay. 1994. Textual tree (prolog) tracer: an experimental evaluation. In *User-Centred Requirements for Software Engineering Environments*. David J. Gilmore, Russel L. Winder, and Françoise Détienne, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 127–141. doi:10.1007/978-3-662-03035-6_11.

[40] Tim Rajan. 1990. Principles for the design of dynamic tracing environments for novice programmers. *Instructional Science*, 19, 4/5, 377–406. http://www.jstor.org/stable/23370485.

[41] Gregory Rosenblatt, Lisa Zhang, William E. Byrd, and Matthew Might. 2019. First-order minikanren representation: great for tooling and search. In *Proceedings of the miniKanren and Relational Programming Workshop* (ICFP Workshop 2019). MiniKanren Workshop, held with ICFP 2019. Association for Computing Machinery.

[42] Dmitry Rozplokhas and Dmitry Boulytchev. 2020. Certified semantics for disequality. In *Proceedings of the 2020 miniKanren and Relational Programming Workshop*. Retrieved July 25, 2025 from http://hdl.handle.net/2047/D20413639.

[43] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitri Boulytchev. 2019. Certified semantics for minikanren. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop*, 80–98. http://nrs.harvard.edu/urn-3:HUL.InstRepos:41307116.

[44] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2020. Certified semantics for relational programming. In *Asian Symposium on Programming Languages and Systems*. Springer, 167–185.

[45] V. Sekovanić and S. Lovrenčić. 2022. Challenges in teaching logic programming. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, 594–598. doi:10.23919/MIPRO55190.2022.9803530.

[46] Ehud Shapiro. 1983. *Algorithmic Program DeBugging*. MIT Press, United States. ISBN: 0262192187.

[47] Dan E. Tamir, Ravi Ananthakrishnan, and Abraham Kandel. 1995. A visual debugger for pure prolog. *Information Sciences - Applications*, 3, 2, 127–147. doi:10.1016/1069-0115(94)00048-7.

[48] Josie Taylor and Ben Du Boulay. 1987. Studying novice programmers: why they may find learning Prolog hard. In *The Computer and Human Development*. J. Rutkowska and C. Crook, (Eds.) Wiley, Chichester.

[49] Markus Triska. 2025. The power of prolog. Online book. Accessed on July 15, 2025. (2025). Retrieved July 15, 2025 from https://www.metalevel.at/prolog.

[50] Mitchell Wand and Dale Vaillancourt. 2004. Relating models of backtracking. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, 54–65. doi:10.1145/1016850.1016861.