A Formal Model and Interactive Visualization of miniKanren Search Semantics

Brysen Pfingsten Jason Hemann

Abstract

Mechanized executable semantics are a valuable tool for implementers and users alike to better understand their programs' behavior. The need is particulary acute in the context of novel logic programming languages with complex search strategies. Few such tools exist for logic-based programming languages however, and existing work only models languages' search behavior at a somewhat coarse-grained level. **In this work, we present the first small-step semantics for a logic language that models interleaving search at the level of individual goal execution.** The model is implemented in PLT Redex and allows users to step through the language's execution at a sub-interleave level.

We also present a visualization tool implemented through interactive JS in the browser with our miniKanren Redex model at its core. This visualizer allows users to input their own miniKanren programs as Racket source code; these programs are transpiled to the language of our model. At each step, users can easily see the evaluation context of their programs, trace goals both to and from the source code and the search tree visualization, and readily understand the history of the computation that led parts of the current state to arise. This not only enables better performance analysis and a unique, visual way of debugging but also allows novice miniKanren programmers to more readily understand the semantics and structure underlying the miniKanren search evolution.

Grammar

$$
\begin{array}{lcl}
p & ::= & (\texttt{prog}\ \Gamma\ e) \\
\Gamma & ::= & ((r_!\ d\ g)\ \ldots) \\
d & ::= & (x_!\ \ldots) \\
s & ::= & () \mid (g\ \sigma) \mid (s \rightarrow s) \mid (s \leftarrow s) \\
 & \mid & ((\top\ \sigma)\ \texttt{+}\ s) \mid (s\ \times\ g) \\
 & \mid & (\texttt{proceed}\ ((r\ t\ \ldots)\ \sigma) \\
 & \mid & (\texttt{delay}\ s) \\
g & ::= & \top \mid (t\ \texttt{=?}\ t) \mid (r\ t\ldots) \\
 & \mid & (g \vee g) \mid (g \wedge g) \mid (\exists\ d\ g) \\
c & ::= & \textbf{natural} \\
x & ::= & \text{(variable-prefix x:)} \\
r & ::= & \text{(variable-prefix r:)} \\
t & ::= & c \mid \textbf{boolean} \mid \textbf{string} \mid \textbf{symbol} \\
 & \mid & x \mid \texttt{empty} \mid (t\ :\ t) \\
\sigma & ::= & (\texttt{state}\ sub\ c) \\
sub & ::= & ((c\ t)\ \ldots)
\end{array}
$$

Figure 1. Abridged grammar of our language.

Evaluation Contexts

$$
\begin{array}{llcl}
\text{Program} & E\Gamma & ::= & (\texttt{prog}\ \Gamma\ \texttt{hole}) \\
\text{Answer Stream} & Ev & ::= & \texttt{hole} \mid ((\top\ \sigma)\ \texttt{+}\ Ev) \\
\text{Search Tree} & Es & ::= & \texttt{hole} \mid (Es \leftarrow s) \mid (s \rightarrow Es) \mid (Es \times g) \\
\text{Composition} & Ex & ::= & E\Gamma[\![Ev[\![Es[\![\texttt{hole}]\!]]\!]]\!]
\end{array}
$$

Figure 2. Evaluation contexts of our language.

Interleaving

At the core of miniKanren's search is the interleave, which ensures fair evaluation by delaying each relation call until it reaches the top of the tree evaluation context. To preserve the visual shape of the tree while enabling this behavior, we implement a *railway model* with directed disjunctions.

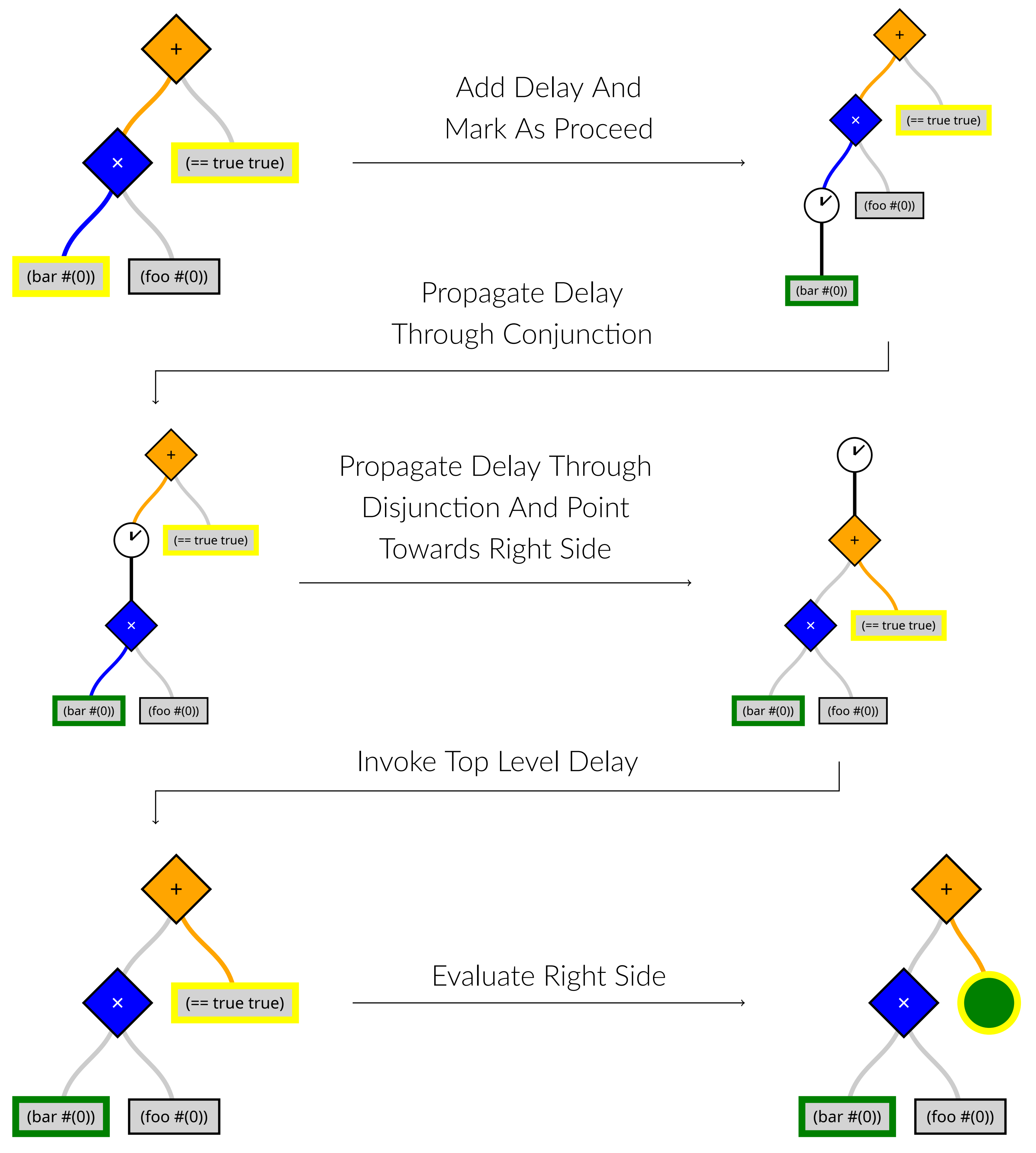

Figure 3. Flow chart demonstrating interleave and railway behavior.

Select Reduction Steps

Distribute State Across Goals to Create Trees

$$
\begin{array}{rcl}
\texttt{Ex}[\![((g_1\ \vee\ g_2)\ \sigma)]\!] & \longrightarrow & \texttt{Ex}[\![((g_1\ \sigma)\ \leftarrow\ (g_2\ \sigma))]\!] \\
\texttt{Ex}[\![((g_1\ \wedge\ g_2)\ \sigma)]\!] & \longrightarrow & \texttt{Ex}[\![((g_1\ \sigma)\ \times\ g_2)]\!]
\end{array}
$$

Railway Model

We show here those reduction steps dealing with left-facing disjunctions and omit those dealing with right-facing disjunctions.

$$
\begin{array}{rcl}
\texttt{Ex}[\![((\texttt{delay}\ s_1)\ \leftarrow\ s_2)]\!] & \longrightarrow & \texttt{Ex}[\![(\texttt{delay}\ (s_1\ \rightarrow\ s_2))]\!] \\
\texttt{Ex}[\![((\top\ \sigma)\ \leftarrow\ s)]\!] & \longrightarrow & \texttt{Ex}[\![((\top\ \sigma)\ \texttt{+}\ s)]\!] \\
\texttt{Ex}[\![(()\ \leftarrow\ s)]\!] & \longrightarrow & \texttt{Ex}[\![s]\!] \\
\texttt{Ex}[\![(((\top\ \sigma)\ \leftarrow\ s_1)\ \leftarrow\ s_2)]\!] & \longrightarrow & \texttt{Ex}[\![((\top\ \sigma)\ \leftarrow\ (s_1\ \leftarrow\ s_2))]\!] \\
\texttt{Ex}[\![((s_1\ \rightarrow\ (\top\ \sigma))\ \leftarrow\ s_2)]\!] & \longrightarrow & \texttt{Ex}[\![((s_1\ \rightarrow\ s_2)\ \rightarrow\ (\top\ \sigma))]\!]
\end{array}
$$

Delays and Proceeds

Relation calls are wrapped with a proceed tag and then a delay tag. Delays at the top level are released and the next time the delayed relation call is encountered it will be expanded.

$$
\begin{array}{rcl}
Ex[\![((r_1\ t\ \ldots)\ \sigma)]\!] & \longrightarrow & Ex[\![(\texttt{delay}\ (\texttt{proceed}\ ((r_1\ t\ \ldots)\ \sigma)))]\!] \\
(\texttt{prog}\ \Gamma\ Ev[\![(\texttt{delay}\ s)]\!] & \longrightarrow & (\texttt{prog}\ \Gamma\ Ev[\![s]\!])
\end{array}
$$

Architecture

Our visualizer is implemented as a web app using D3 to render our trees. Our API is responsible for fulfilling the requests of the user by either sending a cached state if it exists or stepping the program and sending back the corresponding JSON.
You can try it out now at minikanrenredex-prod.shu.edu.

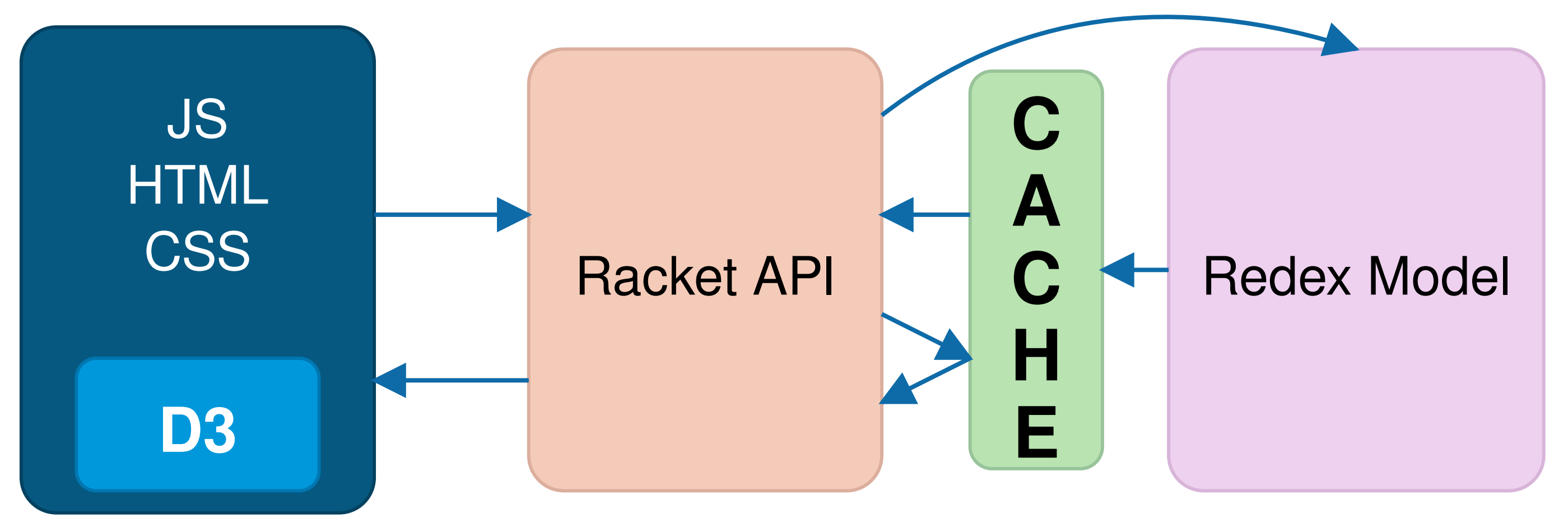

Figure 4. The architecture of our application.