Semantics for EqMLton

Jordan Quinn

July 24, 2025

1 Introduction

1.1 E-Nodes and E-Graphs

Our E-graph implementation will be defined similarly to E-PEGs in Tate's EqSat paper [2]. However, each node is annotated with the possible types it could represent. Therefore, our e-graphs will be 5-tuples, $\langle N, L, C, E, \Gamma \rangle$ where Γ is our typing context.

For nodes n_1 and n_2 in our e-graph, $[\![n_1]\!] = [\![n_2]\!]$ if $L(n_1) = L(n_2)$, $\Gamma \vdash n_1 : \tau$ and $\Gamma \vdash n_2 : \tau$, and, if for $C(n_1) = \langle c_{1,1}, c_{1,2}, \dots \rangle$, $C(n_2) = \langle c_{2,1}, c_{2,2}, \dots \rangle$, $n = |C(n_1)| = |C(n_2)|$, $[\![c_{1,i}]\!] = [\![c_{2,i}]\!]$ for each $i \in \{1, 2, \dots, n\}$.

1.2 Regions

A **region** defines a set of e-graph nodes representing a computation. These region values appear in the CFG-skeleton and essentially represent continuations.

Formally, a region, of an e-graph G, is a three-tuple, $R = \langle N, I, r \rangle$ where N is the set of nodes of G that compose the region, I is a list of parameter nodes in N that represent inputs to our region, and $r \in N$ is the root node for our region that represents the result of the region's computation.

Note that, for region $R = \langle N, I, r \rangle$,

$$\frac{\Gamma \vdash r : \tau \qquad \Gamma \vdash I = \langle i_1, i_2, \dots, i_k \rangle : \langle \tau_1, \tau_2, \dots \tau_k \rangle}{\Gamma \vdash R : \langle \tau_1, \tau_2, \dots, \tau_k \rangle \to \tau}$$

In other words, region R is treated as a continuation that takes a tuple containing arguments for each parameter and returns a value of type τ.

Then, for two regions, $R_1 = \langle N_1, I_1, r_1 \rangle$ and $R_2 = \langle N_2, I_2, r_2 \rangle$ in e-graph $G = \langle N, L, C, E, \Gamma \rangle$, where $N_1 = \langle n_{1,1}, n_{1,2}, \dots \rangle$ and $I_1 = \langle i_{1,1}, i_{1,2}, \dots \rangle$ and similar for N_2, I_2. Then, if $\Gamma \vdash R_1 : \tau$ and $\Gamma \vdash R_2 : \tau$, we know the lists of parameters are the same length and typed in the same order. Then, define L' and e-graph $G' = \langle N, L', C, E, \Gamma \rangle$ such that $L' = L[i_{1,1} \vdash L(i2, 1), i_{1,2} \vdash L(i2, 2), \dots]$. This ensures that the labeling is the same for each parameter list and thus $[\![i_{1,i}]\!]_{G'} = [\![i_{2,i}]\!]_{G'}$ for each $i \in \{1, \dots, |I_1| = |I_2|\}$. Then $[\![R_1]\!]_G = [\![R_2]\!]_G$ if

$[\![r_1]\!]_{G'} = [\![r_2]\!]_{G'}$. In other words, two regions are semantically equivalent if their root nodes are semantically equivalent, ignoring the labels of parameter nodes in our regions.

1.3 CFG Skeleton

The CFG-skeleton is a directed graph that encodes the control-flow decisions taken by a program. It is directly related to MLton's SSA graph, except we are only interested in branch and merge points. Instead of representing computations as statements in the CFG-skeleton, as is done in MLton's SSA, we

The CFG-skeleton is a directed graph where nodes, called blocks, represent branch and merge points in a program, and edges represent control flow transfer. Blocks are related to those of MLton's SSA implementation. They allow for opaque parameters that are referred to within the block. However, where MLton's SSA places computation in statements within each block, our CFG-skeleton only contains control flow information, using regions to represent computations used in our control flow statements. Each block can be defined with one control flow statement. Therefore, it is perhaps more helpful to think of our CFG skeleton as a single root block, $S = b$, and consider it along with a function that returns child blocks, $C(b) = \{b_1, \ldots, b_b\}$, calculated from the control flow statement that comprises each block. Then we can define all blocks in our CFG skeleton, S^*, as the closure of the child function.

$$
\begin{aligned}
b(x_1, \ldots, x_k) ::= \; & \texttt{goto}_{b'} \left(R_1 \left\langle x_{1,1}, \ldots, x_{1,k_1} \right\rangle, R_2 \left\langle x_{2,1}, \ldots, x_{2,k_1} \right\rangle \ldots \right) \\
& \mid \texttt{if } R \texttt{ then goto}_{b_t} \left(R_{1,1} \left\langle x_{1,1,1}, \ldots, x_{1,1,k_{1,1}} \right\rangle, R_{1,2} \ldots \right) \\
& \quad \texttt{else goto}_{b_f} \left(R_{2,1} \left\langle x_{2,1,1}, \ldots, x_{1,k_{2,1}} \right\rangle, R_{2,2} \ldots \right) \\
& \mid \texttt{return } R \left\langle x_1, \ldots, x_{k_1} \right\rangle
\end{aligned}
$$

Here, block b has a single control flow statement that defines outgoing edges. In `goto`, we have one outgoing edge from b to b'. We are passing arguments computed by evaluating regions R_1, R_2, etc. on their corresponding argument lists, $\langle x_{1,1}, \ldots, x_{1,k'} \rangle$ for R_1, etc. where each x is a parameter in $\{x_1, \ldots, x_k\}$. Similar is done to show edges for the other control flow statements. Note that `if` is used as an example for notation; we will solely use match according to the next subsection.

1.3.1 Pattern Matching

The primary control flow consideration is handling pattern matching in our CFG skeleton. We consider three options: tagged tuples, first evaluation, and check-first evaluation.

- **Tagged Tuple Pattern Matching**

In this variation, we consider each constructor a tagged tuple. The first value in our tuple indicates which constructor we are using, and the following values are parameters for that constructor. We then would represent a block with pattern-matching control flow as follows:

$$
\begin{aligned}
b(x_1, \ldots, x_k) ::= \texttt{match}\ & R\langle x_{1,1}, \ldots, x_{1,k'}\rangle \\
& \texttt{'1}\ \Rightarrow \texttt{goto}_{b_1}\left(R_{1,1}\left\langle x_{1,1,1}, \ldots, x_{1,1,k_{1,1}}\right\rangle, R_{1,2}\ldots\right) \\
& \texttt{'2}\ \Rightarrow \texttt{goto}_{b_2}\left(R_{2,1}\left\langle x_{2,1,1}, \ldots, x_{2,1,k_{2,1}}\right\rangle, R_{2,2}\ldots\right) \\
& \vdots \\
& \texttt{'n}\ \Rightarrow \texttt{goto}_{b_n}\left(R_{n,1}\left\langle x_{n,1,1}, \ldots, x_{n,1,k_{n,1}}\right\rangle, R_{n,2}\ldots\right)
\end{aligned}
$$

Where the left-hand side of each '$\Rightarrow$', `'label`, is a label describing a constructor for τ when $\Gamma \vdash R : \tau$. The right-hand side of the '$\Rightarrow$' potentially describes naming parts of each constructor or passing additional parameters needed in b_i.

- **First Evaluation Pattern Matching**

  Here, we take advantage of the fact that this is a theoretical representation of a computation. We define our match to pass control onto the first '$\Rightarrow$' that succeeds in evaluating all of its arguments. These arguments are the same as in tagged tuple pattern matching. However, we specially define match to keep trying each case until all regions evaluate to a non-$\bot$ value.

$$
\begin{aligned}
b(x_1, \ldots, x_k) ::= \texttt{match}\ & R\langle x_{1,1}, \ldots, x_{1,k'}\rangle \\
& \Rightarrow \texttt{goto}_{b_1}\left(R_{1,1}\left\langle x_{1,1,1}, \ldots, x_{1,1,k_{1,1}}\right\rangle, R_{1,2}\ldots\right) \\
& \Rightarrow \texttt{goto}_{b_2}\left(R_{2,1}\left\langle x_{2,1,1}, \ldots, x_{2,1,k_{2,1}}\right\rangle, R_{2,2}\ldots\right) \\
& \vdots \\
& \Rightarrow \texttt{goto}_{b_n}\left(R_{n,1}\left\langle x_{n,1,1}, \ldots, x_{n,1,k_{n,1}}\right\rangle, R_{n,2}\ldots\right)
\end{aligned}
$$

  Note that here, we do not need the original R region in our match, and we can rely on our R_is to construct our operand for us.

- **Check First Pattern Matching**

  This differs from the first two in that it first checks a predicate region, R_{p_i}, for each case to ensure that it can be deconstructed according to the pattern in the original program. The RHS is the same as the previous two versions.

$$
\begin{aligned}
&b(x_1, \ldots, x_k) ::= \\
&\quad \texttt{match}\, R\left\langle x_{1,1}, \ldots, x_{1,k'}\right\rangle \\
&\qquad R_{p_1}\left\langle x_{1,p,1}, \ldots, x_{1,p,k_{1,p}}\right\rangle \Rightarrow \texttt{goto}_{b_1}\left(R_{1,1}\left\langle x_{1,1,1}, \ldots, x_{1,1,k_{1,1}}\right\rangle, R_{1,2} \ldots\right) \\
&\qquad R_{p_2}\left\langle x_{2,p,1}, \ldots, x_{2,p,k_{2,p}}\right\rangle \Rightarrow \texttt{goto}_{b_2}\left(R_{2,1}\left\langle x_{2,1,1}, \ldots, x_{2,1,k_{2,1}}\right\rangle, R_{2,2} \ldots\right) \\
&\qquad \vdots \\
&\qquad R_{p_n}\left\langle x_{n,p,1}, \ldots, x_{n,p,k_{n,p}}\right\rangle \Rightarrow \texttt{goto}_{b_n}\left(R_{n,1}\left\langle x_{n,1,1}, \ldots, x_{n,1,k_{n,1}}\right\rangle, R_{n,2} \ldots\right)
\end{aligned}
$$

Note that here, we do not need the original R region in our match, and we can rely on our R_{p_i}s to construct our operand for us.

1.4 Equality Saturation for MLton

Our IR used for EqSat in MLton will then be a two-tuple, notably $R = \langle S, G\rangle$ where S is a CFG-skeleton and G is a PEG-style E-graph, likely derivative of FEG.

2 Formal Semantics

2.1 E-Graphs and E-Nodes

The semantics of an E-node is defined similarly to in FPEG [1]. That being, for E-Node n,

$$[\![n]\!] = L(n)[\![C(n)]\!]$$

An E-graph, however, does not have a specific semantic meaning.

2.2 Regions

We define the semantics of a region, $R = \langle N, I, r\rangle$, with a list of arguments to R, $X = \langle x_1, \ldots, x_n\rangle$ similarly to the semantics of E-graphs given by Tate, et al. [2]. Notably, we place the arguments passed to the region, which share a type with the list of parameters, $\Gamma(X) \sim \Gamma(I)$, into the e-graph as constant nodes that replace the parameter nodes. This is done via $r' = r[i_1 \mapsto [\![x_1]\!], \ldots, i_n \mapsto [\![x_n]\!]]$. Then, we say that the semantics of $R(X) = R\langle x_1, \ldots, x_n\rangle$ is $[\![R(X)]\!] = [\![r']\!]$. Note that $R(X) \equiv R\,X \equiv R\langle x_1, \ldots, x_n\rangle$ for convenience.

We define the semantics of a region, $R = \langle N, r\rangle$

2.3 CFG Skeleton

Our CFG can be defined using the grammar in figure 1. However, it is useful to think of the set of all block labels, bs. In this grammar, the list of arguments, X, to block b, is described by the list of types in T.

$$
\begin{aligned}
B ::= \;& \langle b, X, T, t \rangle \\
b ::= \;& \mathit{block\_label} \\
T ::= \;& \langle \tau_1, \ldots, \tau_n \rangle \\
t ::= \;& \texttt{goto}_b(R_1, \ldots, R_n) \\
\mid \;& \texttt{call}_f(R_1, \ldots, R_n) \Rightarrow b_{ok}; b_{err} \\
\mid \;& \texttt{raise}(R_1, \ldots, R_n) \\
\mid \;& \texttt{return}(R_1, \ldots, R_n) \\
\mid \;& \texttt{match}(R_p; \\
& \quad c_1 \Rightarrow b_1, \\
& \quad \vdots \\
& \quad c_n \Rightarrow b_n \\
& \quad [; \Rightarrow b_d]) \\
R ::= \;& \mathit{region\_label} \\
X ::= \;& \langle x_1, \ldots, x_n \rangle \\
x ::= \;& \mathit{argument\_label} \\
c ::= \;& \mathit{constructor\_label}
\end{aligned}
$$

Figure 1: Grammar for a CFG

Type	**Meta-Var**	**Purpose**
block	b	CFG block
arg	x	Argument to a block or function
region	R	Region in the E-graph
value	v	An SML value, primitive or constructor
return	r	`Ok`/`Err` Ok or exceptional return
store	S	Store containing variable, value pairs

Figure 2: Table of metavariables used in semantics

`true`

`false`

Figure 3: Semantic-level atomics

$\texttt{block}(b)$	$=$ choose $B = \langle b', X, T, t\rangle \in bs$ such that $b \equiv b'$
$\texttt{sig}(b)$	$= \texttt{block}(b)_T$
$\texttt{params}(b)$	$= \texttt{block}(b)_X : \texttt{sig}(b)$
$\texttt{children}(b)$	$= \{b' \in \texttt{block}(b)_t\}$
$\texttt{is\_prim}(v)$	$=$ `true` if v is a primitive type in MLton else `false`
$\texttt{c\_label}(v)$	$= v$ if $\texttt{is\_prim}(v)$ else v_1

Figure 4: Semantic-level functions

$$\textsc{Return}\ \frac{[\![R]\!]_s = v}{\langle \texttt{return}\, R, s\rangle \Downarrow \texttt{Ok}(v)}$$

$$\textsc{Goto}\ \frac{[\![R_1]\!]_s = v_1 \quad \ldots \quad [\![R_n]\!]_s = v_n \quad \langle \texttt{block}(b)_t, s[\texttt{params}(b) \mapsto \langle v_1, \ldots, v_n\rangle]\rangle \Downarrow r}{\langle \texttt{goto}_b(R_1, \ldots, R_n), s\rangle \Downarrow r}$$

$$\textsc{Raise}\ \frac{[\![R_1]\!]_s = v_1 \quad \ldots \quad [\![R_n]\!]_s = v_n}{\langle \texttt{raise}\ (R_1, \ldots, R_n), s\rangle \Downarrow \texttt{Err}(v_1, \ldots, v_n)}$$

$$\textsc{Call-Ok}\ \frac{[\![R_i]\!]_s = v_i \quad [\![f(\ldots, v_i, \ldots)]\!]_s = \texttt{Ok}(v) \quad \langle b, \langle x : \tau\rangle, t\rangle = \texttt{block}(b_{ok}) \quad \langle t, s[x \mapsto v]\rangle \Downarrow r}{\langle \texttt{call}_f(\ldots, R_i, \ldots) \Rightarrow b_{ok}; b_{err}, s\rangle \Downarrow r}$$

$$\textsc{Call-Err}\ \frac{[\![R_i]\!]_s = v_i \quad [\![f(\ldots, v_i, \ldots)]\!]_s = \texttt{Err}(v) \quad \langle b, \langle x : \tau\rangle, t\rangle = \texttt{block}(b_{err}) \quad \langle t, s[x \mapsto v]\rangle \Downarrow r}{\langle \texttt{call}_f(\ldots, R_i, \ldots) \Rightarrow b_{ok}; b_{err}, s\rangle \Downarrow r}$$

$$\textsc{Match-Case}\ \frac{[\![R_p]\!]_s = v \qquad [\![c_i]\!] \sim [\![\texttt{c\_label}(v)]\!] \qquad \langle b, \langle x_1, x_2, \ldots, x_n\rangle, T, t\rangle = \texttt{block}(b) \quad \langle b_i, s[x_1 \mapsto v_2, x_2 \mapsto v_3, \ldots, v_{n+1}]\rangle \Downarrow r}{\langle \texttt{match}(R_p; \ldots, c_i \Rightarrow b_i, \ldots; \ldots), s\rangle \Downarrow r}$$

$$\textsc{Match-Default}\ \frac{[\![R_p]\!]_s = v_p \quad \forall_{i \in \{1,\ldots,n\}} [\![c_i]\!] \not\sim [\![\texttt{c\_label}(v_p)]\!] \qquad \langle b_d, \langle\rangle, \langle\rangle, t\rangle = \texttt{block}(b_d) \qquad \langle t, s\rangle \Downarrow r}{\langle \texttt{match}(R_p; c_1 \Rightarrow \ldots, \ldots, c_n \Rightarrow \ldots; \Rightarrow b_d), s\rangle \Downarrow r}$$

Figure 5: Semantics for CFG skeleton

In 5, there are a few additional considerations. First, $[\![R\,X]\!]_s$ is the same as $[\![R\,V]\!]$ where $V = \langle s(x_1), \ldots, s(x_n)\rangle$. Then, Match-Case defines the semantics for when we successfully match based on the constructor while Match-Default covers when we do not match on the constructor and execute the default case. Note that we can assume there will be a default case when necessary as we are translating from a well-formed MLton SSA program.

An important consideration is function arguments and globals. These are available to every block in the CFG regardless of where they appear. As such, we start with each function argument and global in our initial store, s.

2.4 CFG Properties

In our CFG skeleton, if we consider each `goto` assignment, as it saves this to the store, we can consider our CFG skeleton to have the dominance property, where each use of a variable is dominated by its definition.

References

[1] Matthew DellaNeve. *Equality Saturation for the MLton Compiler*. Capstone report, Rochester Institute of Technology, 2023.

[2] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Logical Methods in Computer Science*, volume 7, 2011.