

FLAMBDA2 VALIDATOR

1. FLAMBDA2 CORE

$$\begin{aligned}
\text{simple} &::= \text{var} \mid \text{symbol} \mid \text{const} \\
\text{named} &::= \text{simple} \mid \text{prim} \mid P \mid \chi \mid \text{rec_info} \\
\text{exp} &::= \text{named} \mid \text{let } \text{var} = \text{exp}_1 \text{ in } \text{exp}_2 \mid \text{let } (\text{clo } K) = P \text{ in } \text{exp} \\
&\quad \mid \text{let } (\text{code}^\uparrow \text{id}) = \text{code} \text{ in } \text{exp} \mid \text{let } (\text{clo}^\uparrow K) = P \text{ in } \text{exp} \\
&\quad \mid \text{let } (\text{block}^\uparrow b) = \text{block} \text{ in } \text{exp} \mid \text{exp}_1 \text{ where } k \ x = \text{exp}_2 \\
&\quad \mid \text{call}(\kappa) \text{ with } (\text{exp}_\rho, \text{res}_k, \text{exn}_k, \overrightarrow{\text{exp}}) \mid \text{exp}_1 \ \overrightarrow{\text{exp}}_2 \\
&\quad \mid \text{switch } (\text{exp}_1) \ \text{arms} \mid \text{invalid} \\
P &::= \{\text{fns} : (\text{slot}^f * \text{id_exp}) \ \text{map}; \ \text{vals} : (\text{slot}^v * \text{simple}) \ \text{map}\} \\
\text{id_exp} &::= \text{id} \mid \text{exp} \\
\kappa &::= \text{direct } \text{id} \mid \text{indirect} \mid \text{method} \mid \text{c_call} \\
\chi &::= P \mid \text{block}(\text{tag}, \text{mut}, \overrightarrow{\text{exp}}) \mid \dots \\
\text{prim} &::= \text{load}(\text{kind}, \text{mut}, \text{exp}_b) \mid \text{make_block}(\text{kind}, \text{mut}, \overrightarrow{\text{exp}}) \\
&\quad \mid \pi_v(\text{slot}^f) \mid \pi_{f_1}(\text{slot}^{f_2}) \mid \dots
\end{aligned}$$

FIGURE 1. Flambda2 Core Syntax (Abbreviated.)

1.1. Block-based primitives. Blocks correspond to OCaml blocks, which are the word-aligned chunks of memory allocated for representing values¹. Each block has a tag, corresponding to the constructors/field index of a value (e.g. `tag0` is the first constructor of an ADT). The mutability field corresponds to whether the block represents a reference cell. The block-related primitives allows the representation of structs, tuples, lists, and arrays. We plan to support the block-related primitives (`load` from a block and `make_block`) except those related to floating-point valued arrays.

2. REDUCTION STRATEGY

This language has a call-by-value style reduction strategy, as shown in Figure 2. Notice the unusual [LetR] rule—the expression N refers to an expression in the normal form, which may refer to a normalized effectful expression. This rule is *not* analogous to the [ApplyContR] rule, since the lambda abstraction is always implicit in let expressions, ensuring that the

¹For more, see <https://dev.realworldocaml.org/runtime-memory-layout.html>.

$$\begin{array}{c}
\text{LET-}\beta \\
\frac{}{\text{let } x = v \text{ in } e \longrightarrow e[x \setminus v]}
\end{array}
\quad
\begin{array}{c}
\text{LETL} \\
\frac{e_1 \longrightarrow e'_1}{\text{let } x = e_1 \text{ in } e_2 \longrightarrow \text{let } x = e'_1 \text{ in } e_2}
\end{array}$$

$$\begin{array}{c}
\text{LETR} \\
\frac{e_2 \longrightarrow e'_2}{\text{let } x = N \text{ in } e_2 \longrightarrow \text{let } x = N \text{ in } e'_2}
\end{array}$$

$$\begin{array}{c}
\text{LET CLO-}\beta \\
\forall x, x \in X[i], \text{let } (\text{clo } X) = K \text{ in } e \longrightarrow e[x \setminus (\pi_1 K[i], K)]
\end{array}$$

$$\begin{array}{c}
\text{LET STATIC CLO-}\beta \\
\forall x, x \in X^\uparrow, \text{let } (\text{clo } X^\uparrow) = K \text{ in } e \longrightarrow e[x \setminus (x, K)]
\end{array}$$

$$\begin{array}{c}
\text{LET CODE-}\beta \\
\text{let } (\text{code}^\uparrow f(x, \rho, \text{res}_k, \text{exn}_k)) = e_1 \text{ in } e_2 \longrightarrow e_2[f \setminus \lambda(x, \rho, \text{res}_k, \text{exn}_k)].e_1
\end{array}$$

$$\begin{array}{c}
\text{LET CONT-}\beta \\
\frac{}{e_1 \text{ where } k \overrightarrow{\text{args}} = e_2 \longrightarrow e_1[k \setminus \lambda \overrightarrow{\text{args}}.e_2]}
\end{array}
\quad
\begin{array}{c}
\text{APPLY CONT R} \\
\frac{\overrightarrow{\text{args}} \longrightarrow \overrightarrow{\text{args}}'}{v \overrightarrow{\text{args}} \longrightarrow v \overrightarrow{\text{args}}'}
\end{array}$$

$$\begin{array}{c}
\text{APPLY CONT L} \\
\frac{k \longrightarrow k'}{k \overrightarrow{\text{args}} \longrightarrow k' \overrightarrow{\text{args}}}
\end{array}
\quad
\begin{array}{c}
\text{APPLY CONT-}\beta \\
(\lambda x. e) v \longrightarrow e[x \setminus v]
\end{array}$$

$$\begin{array}{c}
\text{APPLY R} \\
\frac{\overrightarrow{\text{args}} \longrightarrow \overrightarrow{\text{args}}'}{\text{call}(e) \text{ with } (\rho, \text{res}_k, \text{exn}_k, \overrightarrow{\text{args}}) \longrightarrow \text{call}(e) \text{ with } (\rho, \text{res}_k, \text{exn}_k, \overrightarrow{\text{args}}')}
\end{array}$$

$$\begin{array}{c}
\text{APPLY L} \\
\frac{e \longrightarrow e'}{\text{call}(e) \text{ with } (\rho, \text{res}_k, \text{exn}_k, \overrightarrow{\text{args}}) \longrightarrow \text{call}(e') \text{ with } (\rho, \text{res}_k, \text{exn}_k, \overrightarrow{\text{args}})}
\end{array}$$

$$\begin{array}{c}
\text{APPLY-}\beta \\
\text{call}(\text{direct}(\lambda(x, \rho, \text{res}_k, \text{exn}_k).e)) \text{ with } (K, \vec{v}, k_r, k_e) \longrightarrow e[\rho \setminus K][x \setminus \vec{v}][\text{res}_k \setminus k_r][\text{exn}_k \setminus k_e]
\end{array}$$

$$\begin{array}{c}
\text{SWITCH} \\
\frac{e \longrightarrow e'}{\text{switch}(e) \text{ arms} \longrightarrow \text{switch}(e') \text{ arms}}
\end{array}
\quad
\begin{array}{c}
\text{SWITCH-}\beta \\
\text{switch}(v)[x \mapsto e] \longrightarrow e[x \setminus v]
\end{array}$$

$$\overrightarrow{\text{args}} \longrightarrow \overrightarrow{\text{args}}' := \forall a, a \in \overrightarrow{\text{args}}. \exists a', a \longrightarrow^* a' \wedge a \in \overrightarrow{\text{args}}'$$

FIGURE 2. Evaluation Rules for Flambda2 Core

“lefthand-side” of the application is always a value. This is necessary because for the case of several effectful expressions (such as a print statement), inlining the occurrence of the expression multiple times will be behaviorally different from the original expression.

The [Apply- β] rule describes the case when the callee is a lambda expression, and the argument is fully evaluated. The expression is beta-reduced, then the resulting value get passed on as an argument to either the return or

exception continuation, depending on whether or not the expression throws an exception.

3. REWRITE RULES

FLATTENMATCH

$$\text{switch } (\text{switch } (e_1) [A \mapsto e_2 : B | ..]) [B \mapsto e'_2 | ..] \longrightarrow \text{switch } (e_1) [A \mapsto e'_2 [B \setminus e_2] | ..]$$

4. FEATURES

A wishlist of desirable inlining/semantic features to support for the validator.

4.1. Inlining.

- function calls
- recursive functions
- inlining (direct calls, within same function)
- cross-module inlining
- low-priority: locals

4.2. Semantics.

- mutable state
- exceptions
- effects (printing, etc.)
- external calls

4.3. Primitives evaluation.

- arithmetic evaluation: commutative and associative laws for arithmetic? It is likely that the commutative/associative laws are not necessary for the simplifier

TODO: Refactor [simplify_primitive].

Environments

$$r ::= () \mid (r_1, r_2, \dots) \mid \pi_i r \mid \langle \rho, \lambda \rho k_r k_e x.e \rangle$$

$$c ::= \langle \rho, \lambda x.e \rangle$$

$$\rho ::= \bullet \mid \rho, x \mapsto r \mid \rho, k \mapsto c \mid \rho, X \mapsto K \mid \rho, X^\uparrow \mapsto K$$

LET- β

$$\langle \text{let } x = v \text{ in } e, \rho \rangle \longrightarrow \langle e, \rho [x \mapsto v] \rangle$$

LETL

$$\frac{\langle e_1, \rho \rangle \longrightarrow \langle e'_1, \rho' \rangle}{\langle \text{let } x = e_1 \text{ in } e_2, \rho \rangle \longrightarrow \langle \text{let } x = e'_1 \text{ in } e_2, \rho' \rangle}$$

LETCLO- β

$$\langle \text{let (clo } X) = K \text{ in } e, \rho \rangle \longrightarrow \langle e, \rho [X \mapsto K] \rangle$$

LETSTATICCLO- β

$$\langle \text{let (clo}^\uparrow X^\uparrow) = K \text{ in } e, \rho \rangle \longrightarrow \langle e, \rho [X^\uparrow \mapsto K] \rangle$$

LETCODE- β

$$\langle \text{let (code}^\uparrow id) = \lambda \rho k_r k_e x.e_1 \text{ in } e_2, \rho \rangle \longrightarrow \langle e, \rho [id \mapsto \langle \rho, \lambda \rho k_r k_e x.e_1 \rangle] \rangle$$

LETCONT- β

$$\langle e_1 \text{ where } k \overrightarrow{args} = e_2, \rho \rangle \longrightarrow \langle e_1, \rho [k \mapsto \langle \rho, \lambda \overrightarrow{args}.e_2 \rangle] \rangle$$

APPLYCONTR

$$\frac{\langle \overrightarrow{args}, \rho \rangle \longrightarrow \langle \overrightarrow{args}', \rho' \rangle}{\langle v \overrightarrow{args}, \rho \rangle \longrightarrow \langle v \overrightarrow{args}', \rho' \rangle}$$

APPLYCONTL

$$\frac{\langle k, \rho \rangle \longrightarrow \langle k', \rho' \rangle}{\langle k \overrightarrow{args}, \rho \rangle \longrightarrow \langle k' \overrightarrow{args}, \rho' \rangle}$$

APPLYCONT- β

$$\frac{\rho(k) = \langle \rho', \lambda x.e \rangle}{\langle k v, \rho \rangle \longrightarrow \langle e [x \setminus v], \rho' \rangle}$$

APPLYR

$$\frac{\langle \overrightarrow{args}, \rho \rangle \longrightarrow \langle \overrightarrow{args}', \rho' \rangle}{\langle \text{call}(\kappa) \text{ with } (exp_\rho, res_k, exn_k, \overrightarrow{args}), \rho \rangle \longrightarrow \langle \text{call}(\kappa) \text{ with } (exp_\rho, res_k, exn_k, \overrightarrow{args}'), \rho' \rangle}$$

APPLYL

$$\frac{\langle exp_\rho, \rho \rangle \longrightarrow \langle exp'_\rho, \rho' \rangle}{\langle \text{call}(\kappa) \text{ with } (exp_\rho, res_k, exn_k, \overrightarrow{args}), \rho \rangle \longrightarrow \langle \text{call}(\kappa) \text{ with } (exp'_\rho, res_k, exn_k, \overrightarrow{args}), \rho' \rangle}$$

APPLY- β

$$\frac{\rho(id) = \langle \rho'', \lambda P k_r k_e x.e \rangle}{\langle \text{call}(\text{direct } id) \text{ with } (\rho', res_k, exn_k, \overrightarrow{exp}), \rho \rangle \longrightarrow \langle e [P \setminus \rho'] [k_r \setminus res_k] [k_e \setminus exn_k] [x \setminus \overrightarrow{exp}], \rho'' \rangle}$$

SWITCH

$$\frac{\langle e, \rho \rangle \longrightarrow \langle e', \rho' \rangle}{\langle \text{switch } (e) \text{ arms}, \rho \rangle \longrightarrow \langle \text{switch } (e') \text{ arms}, \rho' \rangle}$$

SWITCH- β

$$\langle \text{switch } (v) \{x \mapsto e\}, \rho \rangle \longrightarrow \langle e [x \setminus v], \rho \rangle$$

$$\overrightarrow{args} \longrightarrow \overrightarrow{args}' := \forall a, a \in \overrightarrow{args}. \exists a', a \longrightarrow^* a' \wedge a \in \overrightarrow{args}'$$

FIGURE 3. Evaluation Rules for Flambda2 Core (with environments)