VALIDATOR FOR FLAMBDA2 SIMPLIFIER

1. Flambda2 Core

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

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 values[1]. 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

[1] For more, see `https://dev.realworldocaml.org/runtime-memory-layout.html`.

Environments

$$r ::= () \mid (r_1,\, r_2,\, \cdots) \mid \pi_i\ r \mid \langle \rho,\, \lambda k\, x.K \rangle$$
$$c ::= \langle \rho,\, \lambda x.K \rangle$$
$$\rho ::= \bullet \mid \rho,\, x \mapsto r \mid \rho,\, k \mapsto c$$

Let-β
$$\mathsf{let}\ x\ =\ v\ \mathsf{in}\ e \longrightarrow e\,[x \setminus v]$$

LetL
$$\frac{e_1 \longrightarrow e_1'}{\mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ x\ =\ e_1'\ \mathsf{in}\ e_2}$$

LetR
$$\frac{e_2 \longrightarrow e_2'}{\mathsf{let}\ x\ =\ N\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ x\ =\ N\ \mathsf{in}\ e_2'}$$

LetClo-β
$$\forall x\ i, x = X[i],\ \mathsf{let}\ (\mathsf{clo}\ X)\ =\ K\ \mathsf{in}\ e \longrightarrow e\,[x \setminus (\pi_1\ K[i], K)]$$

LetStaticClo-β
$$\forall x, x \in X^{\uparrow},\ \mathsf{let}\ (\mathsf{clo}\ X^{\uparrow})\ =\ K\ \mathsf{in}\ e \longrightarrow e\,[x \setminus (x, K)]$$

LetCode-β
$$\mathsf{let}\ (\mathsf{code}^{\uparrow}\ f\ (x,\, \rho,\, res_k,\, exn_k))\ =\ e_1\ \mathsf{in}\ e_2 \longrightarrow e_2\ [f \setminus \lambda(x,\, \rho,\, res_k,\, exn_k)].e_1$$

LetCont-β
$$e_1\ \mathsf{where}\ k\ \overrightarrow{args}\ =\ e_2 \longrightarrow e_1\ [k \setminus \lambda\ \overrightarrow{args}.\, e_2]$$

ApplyContR
$$\frac{\overrightarrow{args} \longrightarrow \overrightarrow{args}'}{v\ \overrightarrow{args} \longrightarrow v\ \overrightarrow{args}'}$$

ApplyContL
$$\frac{k \longrightarrow k'}{k\ \overrightarrow{args} \longrightarrow k'\ \overrightarrow{args}}$$

ApplyCont-β
$$(\lambda\ x.\, e)\ v \longrightarrow e\ [x \setminus v]$$

ApplyR
$$\frac{\overrightarrow{args} \longrightarrow \overrightarrow{args}'}{\mathsf{call}(e)\ \mathsf{with}\ (\rho,\, res_k,\, exn_k,\, \overrightarrow{args}) \longrightarrow \mathsf{call}(e)\ \mathsf{with}\ (\rho,\, res_k,\, exn_k,\, \overrightarrow{args}')}$$

ApplyL
$$\frac{e \longrightarrow e'}{\mathsf{call}(e)\ \mathsf{with}\ (\rho,\, res_k,\, exn_k,\, \overrightarrow{args}) \longrightarrow \mathsf{call}(e')\ \mathsf{with}\ (\rho,\, res_k,\, exn_k,\, \overrightarrow{args})}$$

Apply-β
$$\mathsf{call}(\mathsf{direct}(\lambda\ (x,\, \rho,\, res_k,\, exn_k).\, e))\ \mathsf{with}\ (K,\, \overrightarrow{v},\, k_r,\, k_e) \longrightarrow e\ [\rho \setminus K]\ [x \setminus \overrightarrow{v}]\ [res_k \setminus k_r]\ [exn_k \setminus k_e]$$

Switch
$$\frac{e \longrightarrow e'}{\mathsf{switch}\ (e)\ arms \longrightarrow \mathsf{switch}\ (e')\ arms}$$

Switch-β
$$\mathsf{switch}\ (v)\ [x \mapsto e] \longrightarrow e\ [x \setminus v]$$

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

Figure 2. Evaluation Rules for Flambda2 Core

expression. This rule is *not* analogous to the [ApplyContR] rule, since the lambda abstraction is always implicit in let expressions, ensuring that the "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 occurence 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

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

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].