VALIDATOR FOR FLAMBDA2 SIMPLIFIER

1. Reduction strategy

This language has a call-by-value style reduction strategy. 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 “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.

The let-bindings that bind static code does not get inlined, as shown in [LetCode]. Here, N denotes any code that is not a $\mathsf{Deleted}$ snippet. Any let-bound code with a deleted code block gets erased, as in [LetCodeDeleted], and in place, every code binding that is a newer version of a code gets assigned the code id of the original code (note that this assumes a unique generation of newer versions of code, which is assumed for simplicity for now).

$$\textsc{Let-}\beta \quad \text{let } x = v \text{ in } e_2 \longrightarrow e_2\,[x \setminus v]$$

$$\textsc{LetL} \quad \frac{e_1 \longrightarrow e_1'}{\text{let } x = e_1 \text{ in } e_2 \longrightarrow \text{let } x = e_1' \text{ in } e_2}$$

$$\textsc{LetR} \quad \frac{e_2 \longrightarrow e_2'}{\text{let } x = N \text{ in } e_2 \longrightarrow \text{let } x = N \text{ in } e_2'}$$

$$\textsc{LetCodeDeleted} \quad \text{let (code } x) = \text{Deleted in } e \longrightarrow e$$

$$\textsc{LetCodeNewer} \quad \text{let (code (newer\_of } x)\ x') = e_1 \text{ in } e_2 \longrightarrow \text{let (code } x) = e_1\,[x' \setminus x] \text{ in } e_2\,[x' \setminus x]$$

$$\textsc{LetStaticClo} \quad \frac{e_2 \longrightarrow e_2'}{\text{let (code } \mathcal{K}^{\uparrow}) = e_1 \text{ in } e_2 \longrightarrow \text{let (code } \mathcal{K}^{\uparrow}) = e_1 \text{ in } e_2'}$$

$$\textsc{LetClo} \quad \text{let (code } \mathcal{K}) = e_1 \text{ in } e_2 \longrightarrow \text{let (code } \mathcal{K}^{\uparrow}) = e_1^{\uparrow} \text{ in } e_2\,[\mathcal{K} \setminus \mathcal{K}^{\uparrow}]$$

$$\textsc{LetCode} \quad \frac{e_2 \longrightarrow e_2'}{\text{let (code } x) = N \text{ in } e_2 \longrightarrow \text{let (code } x) = N \text{ in } e_2'}$$

$$\textsc{LetCont-}\beta \quad e_1 \text{ where } k\ \overrightarrow{args} = e_2 \longrightarrow e_1\,[k \setminus \lambda\,\overrightarrow{args}.\,e_2]$$

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

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

$$\textsc{ApplyCont-}\beta \quad (\lambda\, x.\, e)\ v \longrightarrow e\,[x \setminus v]$$

$$\textsc{ApplyR} \quad \frac{\overrightarrow{args} \longrightarrow \overrightarrow{args}'}{e\ res_k\ exn_k\ \overrightarrow{args} \longrightarrow e\ res_k\ exn_k\ \overrightarrow{args}'}$$

$$\textsc{ApplyL} \quad \frac{e \longrightarrow e'}{e\ res_k\ exn_k\ \overrightarrow{args} \longrightarrow e'\ res_k\ exn_k\ \overrightarrow{args}}$$

$$\textsc{Apply-}\beta \quad (\lambda\ \overrightarrow{x}.\, e)\ res_k\ exn_k\ \overrightarrow{v} \longrightarrow e\,[x \setminus \overrightarrow{v}] \hookrightarrow res_k / exn_k$$

$$\textsc{Switch} \quad \frac{e \longrightarrow e'}{\text{switch } (e)\ arms \longrightarrow \text{switch } (e')\ arms}$$

$$\textsc{Switch-}\beta \quad \text{switch } (v)\ [x \mapsto e] \longrightarrow e\,[x \setminus v]$$

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

LetR

2. Rewrite Rules

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

3. Features

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

3.1. Inlining.

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

3.2. Semantics.

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

3.3. Primitives evaluation.

- arithmetic evaluation: commutative and associative laws for arithmetic? It is likely that the commutative/associative laws are not necessary for the simplifier
- block-based primitives (makeblock, loading from block) The blocks have a tag, corresponding to the constructors (i.e. tag0 is the first constructor) values either are immediate tags or blocks Mutability corresponds to reference cells

  Being able to treat the block-related primitives will resolve supporting the structures below (except for arrays, which have a tricky case involving storing floating-point values. See floating-point valued array optimization)

TODO: Refactor [simplify_primitive].

3.4. Supported structures.

- structs
- tuples
- lists
- arrays