VALIDATOR FOR FLAMBDA2 SIMPLIFIER

1. Core language
2. Reduction strategy

Let-β

$$\mathsf{let}\ x\ =\ v\ \mathsf{in}\ e_2 \longrightarrow e_2\ [x \setminus v]$$

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

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

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

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

ApplyL

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

Apply-β

$$(\lambda\ x.\ e)\ res_k\ exn_k\ \overrightarrow{args} \longrightarrow e\ [x \setminus \overrightarrow{args}]$$

Switch

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

Switch-β

$$\mathsf{switch}\ (e_1)\ [x \mapsto e_2] \longrightarrow e_2\ [x \setminus e_1]$$

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

LetR, Apply

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.

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?
- normalizing other primitives

TODO: Refactor [simplify_primitive].

4.4. **Supported structures.**

- structs
- tuples
- lists
- arrays