

VALIDATOR FOR FLAMBDA2 SIMPLIFIER

1. CORE LANGUAGE

2. REDUCTION STRATEGY

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

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

$$\begin{array}{c} \text{LETCONT-}\beta \\ e_1 \text{ where } k \overrightarrow{args} = e_2 \longrightarrow e_1 [k \setminus \lambda \overrightarrow{args}. e_2] \end{array} \qquad \begin{array}{c} \text{APPLYCONTR} \\ \frac{\overrightarrow{args} \longrightarrow \overrightarrow{args}'}{v \overrightarrow{args} \longrightarrow v \overrightarrow{args}'} \end{array}$$

$$\begin{array}{c} \text{APPLYCONTL} \\ \frac{k \longrightarrow k'}{k \overrightarrow{args} \longrightarrow k' \overrightarrow{args}} \end{array} \qquad \begin{array}{c} \text{APPLYCONT-}\beta \\ (\lambda x. e) v \longrightarrow e [x \setminus v] \end{array}$$

$$\begin{array}{c} \text{APPLYR} \\ \frac{\overrightarrow{args} \longrightarrow \overrightarrow{args}'}{e \text{ res}_k \text{ exn}_k \overrightarrow{args} \longrightarrow e \text{ res}_k \text{ exn}_k \overrightarrow{args}'} \end{array}$$

$$\begin{array}{c} \text{APPLYL} \\ \frac{e \longrightarrow e'}{e \text{ res}_k \text{ exn}_k \overrightarrow{args} \longrightarrow e' \text{ res}_k \text{ exn}_k \overrightarrow{args}} \end{array}$$

$$\begin{array}{c} \text{APPLY-}\beta \\ (\lambda \vec{x}. e) \text{ res}_k \text{ exn}_k \vec{v} \longrightarrow e [x \setminus \vec{v}] \hookrightarrow \text{res}_k / \text{exn}_k \end{array}$$

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

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

LetR

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 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
- 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].

4.4. Supported structures.

- structs
- tuples
- lists
- arrays