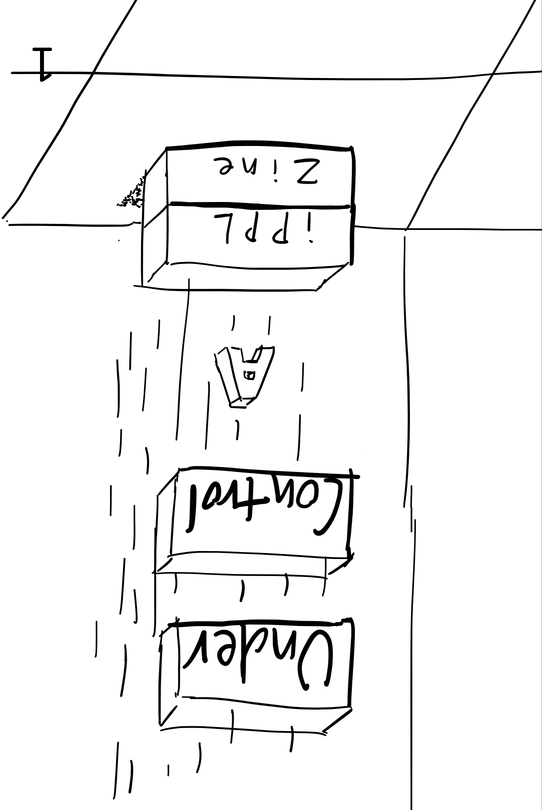

Control Stacks

Control stacks are used for tracking the context of an evaluation step.

Say we have a language:

- all natural numbers are valid expr.
- function def. are valid expr. $(\lambda x. e)$
- function app. are valid expr. $(f\ x)$

Evaluations are done in steps ($\mapsto$) and LHS/RHS should be congruent at all time.

A new instance of an evaluation always starts from an empty stack (ε) where stack/context and expr. are separated with either $\triangleright$ (evaluating) or $\triangleleft$ (returning)

Example

$\varepsilon \triangleright (\lambda x. x \cdot 2)(3)$

Example

Say we have a very simple function that takes in an integer x, and we define the function as:

$\lambda x.\ \overline{case}\ (x, a.a+1, b.b+2)$

Where $\overline{case}(\ell, m.m, n.n)$ expects an $\overline{in_1}e$ or $\overline{in_2}e$ to determine which branch to take where e represents some expression.

Let's pass $\overline{in_1}9$ to ℓ: $\overline{case}(\overline{in_1}9, m.m, n.n)$

It means this $\overline{case}$ will take the first branch and substitute m with 9 (formally: $[9/m]m$)

and $\overline{case}(\overline{in_1}9, m.m, n.n)$ will step to 9

When trying to make sense of $\overline{case}$ and $\overline{in_1}/\overline{in_2}$

→ in the $\overline{case}$ expression: $\overline{case}(\overline{in_1}e, a.a, b.b)$ e will bind to whichever case it go to.

→ $\overline{in_1}/\overline{in_2}$ basically acts as an indicator of where e will bind to (the first or the second)

Execution

Now let's evaluate the following:

$\lambda x.\ \overline{case}\ (x, a.a+1, b.b+2)(\overline{in_2}(1+2))$

① $\mapsto \varepsilon \triangleright \lambda x.\ \overline{case}\ (x, a.a+1, b.b+2)(\overline{in_2}(1+2))$

② $\mapsto (\square(\overline{in_2}(1+2))) \triangleright \lambda x.\ \overline{case}\ (x, a.a+1, b.b+2)$

③ $\mapsto (\square(\overline{in_2}(1+2))) \triangleleft \lambda x.\ \overline{case}\ (x, a.a+1, b.b+2)$

④ $\mapsto (\lambda x.\overline{case}(x, a.a+1, b.b+2)(\square)) \triangleright \overline{in_2}(1+2)$

⑤ $\mapsto (\lambda x.\overline{case}(x, a.a+1, b.b+2)\square; \overline{in_2}\square) \triangleright (1+2)$

⑥ $\mapsto^* (\lambda x.\overline{case}(x, a.a+1, b.b+2)\square; \overline{in_2}\square) \triangleleft 3$

⑦ $\mapsto (\lambda x.\overline{case}(x, a.a+1, b.b+2)\square) \triangleleft \overline{in_2}3$

⑧ $\mapsto \varepsilon \triangleright \overline{case}(\overline{in_2}3, a.a+1, b.b+2)$

⑨ $\mapsto^* \overline{case}(\square, a.a+1, b.b+2) \triangleleft \overline{in_2}3$

⑩ $\mapsto \varepsilon \triangleright 3+2$

⑪ $\mapsto^* \varepsilon \triangleleft 5$

How ???

* "□" represents a hole and it'll only appear on stack side. It's an indication of the expr that's currently being evaluated/returned (▷|◁)

① → Evaluation start from an empty stack

② → Eval of function application requires the function to be evaluated first, the lambda expr gets replaced with a hole

③ → Lambda exp's are values, directly return

④ → Eval the calling arguments

⑤ → Some more computations require here, in the full form (1+2) needs six steps: two for 1, two for + and two for 2

⑥ → After the skipped 6 steps, "3" will be returned to fill the □ from $\overline{in_2}(\dots)$

⑦
⑧ → The evaluated "$\overline{in_2}3$" substitutes "x" in $\overline{case}$

⑨ → Check which branch to take within $\overline{case}$

⑩
⑪ → second branch chosen, eval and return 3+2

Correctness

You can't just write random stuff during evaluations, right?

Some properties have to be maintained when between stack machine state transitions ($\mapsto$ | $\mapsto^*$) (the steps we've taken)

Symbols

- ε empty stack
- $\mapsto$ single step
- $\mapsto^*$ step star
- $k \triangleright e$ evaluating e w/k
- $k \triangleleft v$ returning v to k
- ☆: ε can be generalized to any stack

Completeness

If $e \mapsto^* e'$ and e' is value then $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e'$

ie. if an expression steps to a value, it'll get returned to whatever stack☆ it's evaluated under.

Soundness

If $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e'$ then $e \mapsto^* e'$ and e' is value

ie. if e' can be returned to some stack from evaluation e, then e must be able to step to e' and e' is a value.

WDYM :(

About completeness: take a look at the steps ⑤/⑥ of our example. Since we know 1+2 will eventually step to 3 if it gets evaluated:

$1+2 \mapsto^* 3$

From this,

⑤ $\mapsto (\lambda x.\overline{case}(x, a.a+1, b.b+2)\square; \overline{in_2}\square) \triangleright (1+2)$

⑥ $\mapsto^* (\lambda x.\overline{case}(x, a.a+1, b.b+2)\square; \overline{in_2}\square) \triangleleft 3$

In reverse, from ⑥/⑤, the fact that 3 gets returned guarantees the evaluation of 1+2 eventually step to 3 and 3 is indeed a value. (Soundness)

Pretty cool right? To get to know the full syntax and typing/congruence rules, refer to iPPL course website and PFPL textbook

Thanks!

IMO this control stack mechanism is exactly stepping rules plus context, where you explicitly mark the expression(s) you are evaluating

The correctness properties (completeness | soundness) are like oracles that tells if you have a stepping relation between expressions, you can evaluate them and eventually terminate. (if well-typed)

References

- Typing rules, stack syntax, evaluation rules ccs.neu.edu/~cmartens/Courses/7400-f24
- Safety properties, correctness/soundness prof, control machine definitions Ch.27, Practical Foundations for Programming Languages
- Implementing this in Ocaml! CS3110, Cornell University cs3110.github.io
- Extended version of this Zine ycun.co/control