PCF (refresh)

$$\begin{aligned}
Var, \{z\} &\subset PCF \\
s(e) &\in PCF \text{ for } e \in PCF \\
\text{ifz}\,(e_1, x.e_2) &\in PCF \text{ for } e_1, e_2 \in PCF,\ x \in Var \\
\text{lam}\,(x.e) &\in PCF \text{ for } e \in PCF,\ x \in Var \\
\text{ap}\,(e_1, e_2) &\in PCF \text{ for } e_1, e_2 \in PCF \\
\text{fix}\,(x.e) &\in PCF \text{ for } e \in PCF,\ x \in Var
\end{aligned}$$

Expressions are analysed for consistent types, and evaluation (to what is judged a `val`, textitz, s(e), lam) proceeds based on evaluation rules. While every rule does have just one premise that would require a sub-evaluation, this sub-evaluation is assumed to magically be finished and the outcome known, which makes it unfavourable for implementation.

K machine

The K machine evaluates PCF programs using a control stack instead of context provided by premises in rules. Thereby, a practical implementation can be derived directly from its definition.

Domains and states

- Expressions: $e \in PCF$
- Stack frames:

$$F = \{s(-)\} \cup \{ap(-, e) | e \in PCF\} \cup \{ifz(e_1, x.e_2)(-) | e_1, e_2 \in PCF; x \in Var\}$$

- Stack: $St = F^*$ a list of stack frames, possibly empty (ε)

  My stack grows to the left (and uses (:) cons), whereas Harper's grows to the right.
- Machine state: $S = St \times \{\lhd, \rhd\} \times PCF$

  $\lhd$ indicates use of the topmost stack frame, $\rhd$ indicates *PCF* evaluation.

Transitions

The K machine is based on an evaluation relation on machine states, $\mapsto \in S \times S$:

- Numbers:

$$\begin{aligned}
k \rhd z &\mapsto k \lhd z \\
k \rhd s(e) &\mapsto s(-) : k \rhd e \\
s(-) : k \lhd e &\mapsto k \lhd s(e)
\end{aligned}$$

- Branching:

$$\begin{aligned}
k \rhd ifz(e_1, x.e_2)(e) &\mapsto ifz(e_1, x.e_2)(-) : k \rhd e \\
ifz(e_1, x.e_2)(-) : k \rhd z &\mapsto k \rhd e_1 \\
ifz(e_1, x.e_2)(-) : k \rhd s(e) &\mapsto k \rhd [e/x]\, e_2
\end{aligned}$$

- Functions:

$$\begin{array}{rcl} k \triangleright lam(x.e) & \mapsto & k \triangleleft lam(x.e) \\ k \triangleright ap(e_1, e_2) & \mapsto & ap(-, e_2) : k \triangleright e_1 \\ ap(-, e_2) : k \triangleright lam(x.e) & \mapsto & k \triangleright [e_2/x]\, e \\ k \triangleright fix(x.e) & \mapsto & k \triangleright [fix(x.e)/x]\, e \end{array}$$

Somewhat strangely, functions are applied to unevaluated expression arguments (call by name). Other than that, no surprises.

Start and End The machine evaluates an $e \in PCF$ by starting in state $S_{start} = \varepsilon \triangleright e$ and performing evaluations from $\mapsto$ until end state $\varepsilon \triangleleft e$ with a value e.

Safety

Safety is argued for by a type-like property of stack frames.

Stack frames "expect" a type (obvious definition), and usually "apply" themselves to transform that expected type to another. Stack frames are stacked up with matching types according to this idea:

> If stack k expects τ' and frame f expects τ and yields τ', then stack $f : k$ is well-formed and expects *tau*. The empty stack can expect any type as required.

The stack frame type must match the type of the expression under evaluation at all times.

The proof of safety relates well-formed stacks and matching expressions to the evaluation relation: Shows that the K machine won't get stuck in K machine states without successor (i.e. in a state where stack and expression have a type mismatch) or reach an ill-formed stack.

Induction over machine state transitions proves the *ok* part, the productivity is proved by case analysis on the *ok* rules and the state well-formed-ness (checking all transition cases for the non-final case).

Correctness according to *PCF*

Any *PCF* expression corresponds to a well-formed start state $\varepsilon \triangleright e$, it is safe to evalute *PCF* expressions. Does it do the right thing then? (**correctness**, i.e. (**soundness** and **completeness**))

Complete Any *PCF* expression that evaluates to a value does so with the K machine

Sound Anything that the K machine evaluates to a final state is a *PCF* expression that evaluates to the corresponding final value

For **completeness**, an induction over the evaluation in *PCF* is required, giving rise to a) generalising to any stack instead of the empty one, and b) using an *evaluation dynamics* of *PCF* (i.e. values not evaluations) to be able to apply the induction hypothesis.

We effectively prove (Lemma 28.2): if $e \Downarrow v$ then $k \triangleright e \mapsto^* k \triangleleft v$ for every stack k, and assume the $\Downarrow$ evaluation dynamics relates this to $e \mapsto^* v$.

For **soundness**, a notion of "unravels-to" is introduced to abstract from multistep evaluation: $s \looparrowright e$ will give us that if state s is final then e is a value, and otherwise, a transition to $s' \looparrowright e'$ will lead to a value v i.e. (Lemma 28.3): if $s \mapsto s'$ and $s \looparrowright e, s' \looparrowright e'$ then $e \mapsto^* e'$.

This is argued using another, complicated, judgement $\bowtie$, which effectively describes the stack evaluation and relates it to *PCF* evaluation (Lemma 28.6).