

STORE: DYNAMICS

Alex Kavvos

Reading: PFPL §34.1.2, 34.1.3

Recall that a **store** is a finite map $\mu : \text{Loc} \rightarrow \text{StoreVal}$, and that we write $\mu = \mu' \otimes \{a \mapsto v\}$ to mean that μ maps the location $a \in \text{Loc}$ to the value v (i.e. $\mu(a) \simeq v$), and that μ' is the rest of the store.

The dynamics of stores consist of the following judgements.

$$e \text{ val} \qquad m \parallel \mu \text{ final} \qquad e \mapsto e' \qquad m \parallel \mu \mapsto_{\Sigma} m' \parallel \mu'$$

1 Values and Final States

$e \text{ val}$ is the same as for PCF expressions (but indexed by Σ) with one additional rule:

$$\frac{\text{VAL-CMD}}{\text{cmd}(m) \text{ val}}$$

The new judgement $m \parallel \mu \text{ final}$ states that the command m has finished running, leaving the store in state μ . Only one command is final, viz. the one that returns a *value*:

$$\frac{\text{FINAL-RET} \quad e \text{ val}}{\text{ret}(e) \parallel \mu \text{ final}}$$

2 Transitions

The expression transitions $e \mapsto e'$ are much the same as in PCF. Command transitions:

$$\begin{array}{c} \text{D-GET} \\ \frac{}{\text{get}[a] \parallel \mu \otimes \{a \mapsto e\} \mapsto_{\Sigma, a} \text{ret}(e) \parallel \mu \otimes \{a \mapsto e\}} \end{array} \qquad \begin{array}{c} \text{D-SET-1} \\ \frac{e \mapsto e'}{\text{set}[a](e) \parallel \mu \mapsto_{\Sigma, a} \text{set}[a](e') \parallel \mu} \end{array}$$

$$\begin{array}{c} \text{D-SET} \\ \frac{e \text{ val}}{\text{set}[a](e) \parallel \mu \otimes \{a \mapsto -\} \mapsto_{\Sigma, a} \text{ret}(e) \parallel \mu \otimes \{a \mapsto e\}} \end{array} \qquad \begin{array}{c} \text{D-RET-1} \\ \frac{e \mapsto e'}{\text{ret}(e) \parallel \mu \mapsto_{\Sigma} \text{ret}(e') \parallel \mu} \end{array}$$

$$\begin{array}{c} \text{D-BND-1} \\ \frac{e \mapsto e'}{\text{bnd}(e; x. m) \parallel \mu \mapsto_{\Sigma} \text{bnd}(e'; x. m) \parallel \mu} \end{array}$$

$$\begin{array}{c} \text{D-BND-CMD} \\ \frac{m_1 \parallel \mu \mapsto_{\Sigma} m'_1 \parallel \mu'}{\text{bnd}(\text{cmd}(m_1); x. m_2) \parallel \mu \mapsto_{\Sigma} \text{bnd}(\text{cmd}(m'_1); x. m_2) \parallel \mu'} \end{array}$$

$$\begin{array}{c} \text{D-BND-RET} \\ \frac{e \text{ val}}{\text{bnd}(\text{cmd}(\text{ret}(e)); x. m) \parallel \mu \mapsto_{\Sigma} m[e/x] \parallel \mu} \end{array} \qquad \begin{array}{c} \text{D-DCL-1} \\ \frac{e \mapsto e'}{\text{dcl}(e; a. m) \parallel \mu \mapsto_{\Sigma} \text{dcl}(e'; a. m) \parallel \mu} \end{array}$$

$$\begin{array}{c} \text{D-DCL-2} \\ \frac{e \text{ val} \quad m \parallel \mu \otimes \{a \mapsto e\} \mapsto_{\Sigma, a} m' \parallel \mu' \otimes \{a \mapsto e'\}}{\text{dcl}(e; a. m) \parallel \mu \mapsto_{\Sigma} \text{dcl}(e'; a. m') \parallel \mu'} \end{array} \qquad \begin{array}{c} \text{D-DCL-RET} \\ \frac{e \text{ val} \quad e' \text{ val}}{\text{dcl}(e; a. \text{ret}(e')) \parallel \mu \mapsto_{\Sigma} \text{ret}(e') \parallel \mu} \end{array}$$

The rules **D-DCL-1**, **D-DCL-2**, and **D-DCL-RET** implicitly define the concept of **block structure**. As a result, this language can be implemented using just a stack: there is no heap allocation! (Hence everything is deterministic.)

3 Type safety

We define

$$\mu : \Sigma \stackrel{\text{def}}{=} \forall a \in \Sigma. \exists e. \mu(a) \simeq e \wedge e \text{ val} \wedge \vdash_{\emptyset} e : \text{Nat}$$

Type safety is given by the following two theorems. Both are shown by **simultaneous induction**.

Theorem 1 (Preservation).

1. If $\vdash_{\Sigma} e : \tau$ and $e \mapsto e'$ then $\vdash_{\Sigma} e' : \tau$.
2. If $\vdash_{\Sigma} m \text{ ok}$, $\mu : \Sigma$, and $m \parallel \mu \mapsto_{\Sigma} m' \parallel \mu'$ then $\vdash_{\Sigma} m' \text{ ok}$ and $\mu' : \Sigma$.

Theorem 2 (Progress).

1. If $\vdash_{\Sigma} e : \tau$ then either $e \text{ val}$ or $e \mapsto e'$ for some e' .
2. If $\vdash_{\Sigma} m \text{ ok}$ and $\mu : \Sigma$ then either $m \parallel \mu \text{ final}$ or $m \parallel \mu \mapsto_{\Sigma} m' \parallel \mu'$ for some m', μ' .

4 First-order, mobility, and CBV

Notice that the dynamics of MA have a strong call-by-value flavour. For example, when executing a command $\text{ret}(e)$ the rule **D-RET-1** forces e to be fully evaluated to a numeral before returning it. Similarly, when executing $\text{dcl}(e; a. m)$ the rules **D-DCL-1** and **D-DCL-2** force e to be fully evaluated before assigning it to the store location a .

Furthermore, the judgement $\mu : \Sigma$ in the type safety proof requires that everything in the store be (a) numerical, (b) a value, and (c) typable without referring to anything in the store (the subscript Σ is required to be empty).

It is interesting to contemplate what would happen if these requires were not in place. In fact, the problem becomes evident even without far-reaching changes. Suppose we considered $\text{succ}(e)$ to be a value, irrespective of the status of e . (This would give natural numbers a **lazy** semantics.) Then, consider the command

$$m \stackrel{\text{def}}{=} \text{dcl}(a; \text{zero}. \text{ret}(\text{proc}(x : \text{Nat}) \{ \text{set}[a](x) \}))$$

Executing this in the empty store \emptyset allocates a with zero, and then returns a procedure. The final state is

$$\text{ret}(\text{proc}(x : \text{Nat}) \{ \text{set}[a](x) \}) \parallel \emptyset$$

When this procedure is called, it attempts to store the value of its argument x in a . But this is nonsensical, as a is no longer allocated! Thus, the location a has **escaped the scope** of the declaration $\text{dcl}(a; \text{zero}. -)$. Not knowing what a is, this procedure will generate a **stuck state**: the progress theorem will fail.

Thus, for MA to be type-safe, we must include the rule $\frac{e \text{ val}}{\text{succ}(e) \text{ val}} \text{ VAL-SUCC}$ in the dynamics.

A similar problem occurs if the dynamics allow CBN-like behaviour in $\text{ret}(-)$ or $\text{dcl}(-; -, -)$.

Furthermore, other issues occur if we extend the store to allow non-numerical values. For example, if we allowed expressions of the form $\lambda x. e$, then the body e —which is almost never a value—could contain locations a which would then escape their scope of declaration.

How does the type safety proof work then? The key is the property of **mobility** that natural numbers satisfy.

Lemma 3 (Mobility). If $\vdash_{\Sigma} e : \text{Nat}$ and $e \text{ val}$ then $\vdash_{\emptyset} e : \text{Nat}$.

Mobile data are data that do not refer to locations, and hence can be placed in the store. Mobility does not hold if we omit VAL-SUCC.

We have left the dynamics of the PCF part of MA unspecified. The only restriction is that natural numbers need to be strict, i.e. the rule VAL-SUCC must be included. Otherwisewe are free to do anything we like: our language is **stratified** in a mostly-pure part (PCF) and the command language, so that the first is independent of the second.