

STORE

Alex Kavvos

Reading: PFPL §34.1.1, 34.2

Imperative programs are distinguished from functional programs by the use of a **store** (US English: **memory**). We will present Modernised Algol (MA), an imperative language which extends PCF with a store.

1 Stores

Mathematically, a store can be modelled as a finite partial function

$$\mu : \text{Loc} \rightarrow \text{StoreVal}$$

from a set Loc of **locations** to the set StoreVal of **storable values**. Loc is usually required to be infinite, so that we can always allocate more memory; for example we may pick $\text{Loc} \stackrel{\text{def}}{=} \mathbb{N}$.

The set of storable values determines what can be put in the store. In many languages only first-order data (e.g. integers, booleans, ...) and aggregates thereof (e.g. structs, records, ...) are storable. However, languages like OCaml, Scala and JavaScript have a **higher-order store**, i.e. functions can be stored in memory. In this unit we will focus on a **first-order store**, so we pick $\text{StoreVal} \stackrel{\text{def}}{=} \{v \mid \vdash v : \text{Nat} \wedge v \text{ val}\}$.

Finally, the function μ is **finite**, i.e. its domain $\text{dom}(\mu)$ is a finite set; in other words, we are only allowed to use a finite number of locations at any point in time.

We will 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 (i.e. $a \notin \text{dom}(\mu')$).

2 Commands

In addition to the **terms** of the STLC and PCF, MA will have **commands**. While the purpose of expressions is to evaluate to a value, the purpose of commands will be to **change the store**, before also evaluating to a value.

The syntax chart is that of PCF plus the following extensions.

types	τ	$::=$	Nat $\tau_1 \rightarrow \tau_2$ Cmd	natural numbers (partial) function type unevaluated commands
pre-terms	e	$::=$	\dots $\text{cmd}(m)$	unevaluated command
pre-commands	m	$::=$	$\text{ret}(e)$ $\text{bnd}(e; x. m)$ $\text{dcl}(e; a. m)$ $\text{get}[a]$ $\text{set}[a](e)$	return value sequence allocate fetch location contents set location contents
			$\text{ret } e$ $\text{bind } x \leftarrow e; m$ $\text{decl } a := e \text{ in } m$ $@ a$ $a := e$	

The statics of the language have two sorts of judgment:

$$\Gamma \vdash_{\Sigma} e : \tau$$

$$\Gamma \vdash_{\Sigma} m \text{ ok}$$

Both are parameterised in a finite set $\Sigma \subseteq \text{Loc}$ of locations in use. The first judgement is the usual term typing. The second confirms that m is a well-formed command, using values from the context Γ .

The statics of the language are those of PCF (with the additional Σ inserted everywhere) plus the following rules.

$$\begin{array}{c}
\text{FETCH} \\
\frac{}{\Gamma \vdash_{\Sigma, a} \text{get}[a] \text{ ok}}
\end{array}
\quad
\begin{array}{c}
\text{ASSIGN} \\
\frac{\Gamma \vdash_{\Sigma, a} e : \text{Nat}}{\Gamma \vdash_{\Sigma, a} \text{set}[a](e) \text{ ok}}
\end{array}
\quad
\begin{array}{c}
\text{RET} \\
\frac{\Gamma \vdash_{\Sigma} e : \text{Nat}}{\Gamma \vdash_{\Sigma} \text{ret}(e) \text{ ok}}
\end{array}
\quad
\begin{array}{c}
\text{CMD} \\
\frac{\Gamma \vdash_{\Sigma} m \text{ ok}}{\Gamma \vdash_{\Sigma} \text{cmd}(m) : \text{Cmd}}
\end{array}$$

$$\begin{array}{c}
\text{BIND} \\
\frac{\Gamma \vdash_{\Sigma} e : \text{Cmd} \quad \Gamma, x : \text{Nat} \vdash_{\Sigma} m \text{ ok}}{\Gamma \vdash_{\Sigma} \text{bnd}(e; x. m) \text{ ok}}
\end{array}
\quad
\begin{array}{c}
\text{DECL} \\
\frac{\Gamma \vdash_{\Sigma} e : \text{Nat} \quad \Gamma \vdash_{\Sigma, a} m \text{ ok}}{\Gamma \vdash_{\Sigma} \text{dcl}(e; a. m) \text{ ok}}
\end{array}$$

Commands potentially change the store, and then return a value.

The commands $\text{get}[a]$ and $\text{set}[a](e)$ respectively fetch the value at location a , and assign the value of $e : \text{Nat}$ at location a . Notice that the location needs to be allocated, i.e. included in the subscript.

The command $\text{ret}(e)$ simply returns the value of a natural number expression, without changing the store.

The rule **CMD** says that any command m can be seen as an expression $\text{cmd}(m) : \text{Cmd}$. The command is *not* executed when this term is evaluated, but remains frozen in place. Thus, terms of the form $\text{cmd}(m)$ are values.

The command $\text{bnd}(e; x. m)$ is a **sequencing** construct. It evaluates e until it becomes a value $\text{cmd}(p)$, and then runs p . The value returned by p is then bound to x , and the next command m is run.

$\text{dcl}(e; a. m)$ declares the new location a by assigning the value of term e to it. Notice that the typing of m ensures that a is a valid location (it is included in the subscript). The **scope** of this declaration is the command m , which runs after the allocation. When its execution is complete, the location a gets de-allocated. Thus, this construct creates **block structure**, and hence enforces a **stack discipline** (= a stack suffices to implement it).

3 Examples

To relate Modernised Algol to common programming idioms we may define the following shorthands.

$$\{x \leftarrow m_1; m_2\} \stackrel{\text{def}}{=} \text{bnd}(\text{cmd}(m_1); x. m_2) \quad \{m_1; m_2\} \stackrel{\text{def}}{=} \text{bnd}(\text{cmd}(m_1); _ . m_2) \quad \text{do } e \stackrel{\text{def}}{=} \text{bnd}(e; x. \text{ret}(x))$$

We sometimes write $\{m_1; m_2; \dots; m_n\} \stackrel{\text{def}}{=} \{m_1; \{m_2; \{\dots; m_n\}\}\}$, and similarly if we have bindings.

Armed with these shorthands we can write **conditionals** and **loops** as follows:

$$\begin{aligned}
\text{if } m \text{ then } m_1 \text{ else } m_2 &\stackrel{\text{def}}{=} \{x \leftarrow m; \text{do ifz}(x; \text{cmd}(m_1); _ . \text{cmd}(m_2))\} \\
\text{while } (m) \{m^*\} &\stackrel{\text{def}}{=} \text{do fix}(r : \text{Cmd}. \text{cmd}(\text{if } m \text{ then } (\text{ret zero}) \text{ else } \{m^*; \text{do } r\}))
\end{aligned}$$

A **procedure** is a term $f : \tau \rightarrow \text{Cmd}$. We define

$$\text{proc } (x : \tau) \{m\} \stackrel{\text{def}}{=} \lambda x : \tau. \text{cmd}(m) \quad \text{call } e_1(e_2) \stackrel{\text{def}}{=} \text{do } e_1(e_2)$$

We can then write programs like the following one, which computes the factorial of x .

```

proc (x : nat) {
  decl r := 1 in
  decl a := x in
  {
    while (@a) {
      y <- @r; z <- @a;
      r := y * (x - z + 1);           // r := r * (x - a + 1)
      a := z - 1;                     // a := a - 1
    };
    x <- @r;
    ret x
  }
}

```

The invariant for this loop is $@r = (x - @a)!$, so at the end $@r = x!$.