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

$$\mu : \mathsf{Loc} \rightharpoonup \mathsf{StoreVal}$$

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 $\mathsf{Loc} \stackrel{\text{def}}{=} \mathbb{N}$.

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 $\mathsf{StoreVal} \stackrel{\text{def}}{=} \{v \mid \vdash v : \mathsf{Nat} \wedge v \;\mathsf{val}\}$.

Finally, the function μ is **finite**, i.e. its domain $\mathsf{dom}(\mu)$ 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 $a \in \mathsf{Loc}$ to the value v (i.e. $\mu(a) \simeq v$), and that μ' is the rest of the store (i.e. $a \notin \mathsf{dom}(\mu')$).

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

The statics of the language have two sorts of judgment:

$$\Gamma \vdash_\Sigma e : \tau \qquad\qquad \Gamma \vdash_\Sigma m\ \mathsf{ok}$$

Both are parameterised in a finite set $\Sigma \subseteq \mathsf{Loc}$ 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.

$$\textsc{Cmd}\ \frac{\Gamma \vdash_\Sigma m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathsf{cmd}(m) : \mathsf{Cmd}} \qquad \textsc{Ret}\ \frac{\Gamma \vdash_\Sigma e : \mathsf{Nat}}{\Gamma \vdash_\Sigma \mathsf{ret}(e)\ \mathsf{ok}} \qquad \textsc{Bind}\ \frac{\Gamma \vdash_\Sigma e : \mathsf{Cmd} \qquad \Gamma, x : \mathsf{Nat} \vdash_\Sigma m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathsf{bnd}(e; x.\, m)\ \mathsf{ok}}$$

$$\textsc{Decl}\ \frac{\Gamma \vdash_\Sigma e : \mathsf{Nat} \qquad \Gamma \vdash_{\Sigma,a} m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathsf{dcl}(e; a.\, m)\ \mathsf{ok}} \qquad \textsc{Fetch}\ \frac{}{\Gamma \vdash_{\Sigma,a} \mathsf{get}[a]\ \mathsf{ok}} \qquad \textsc{Assign}\ \frac{\Gamma \vdash_{\Sigma,a} e : \mathsf{Nat}}{\Gamma \vdash_{\Sigma,a} \mathsf{set}[a](e)\ \mathsf{ok}}$$

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

$\mathsf{dcl}(e; a.\, m)$ 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).

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

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

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

3 Examples

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

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

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

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

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

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

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

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 $\texttt{@r} = (\texttt{x} - \texttt{@a})!$, so at the end $\texttt{@r} = \texttt{x}!$.