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

Finally, the function μ is **finite**, i.e. its domain dom(μ) 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.

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

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.

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

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

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

The corresponding command bnd(e; x. m) is a **sequencing** construct. It evaluates e until it becomes a value 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) \quad \{m_1; m_2\} \stackrel{\text{def}}{=} \mathsf{bnd}(\mathsf{cmd}(m_1); \underline{\ }. \, m_2) \quad \mathsf{do} \, e \stackrel{\text{def}}{=} \mathsf{bnd}(e; x. \, \mathsf{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:

```
if m then m_1 else m_2 \stackrel{\text{def}}{=} \{x \leftarrow m; \operatorname{doifz}(x; \operatorname{cmd}(m_1); \_.\operatorname{cmd}(m_2))\}
while (m)\{m^*\} \stackrel{\text{def}}{=} \operatorname{dofix}(r : \operatorname{Cmd}.\operatorname{cmd}(\operatorname{if} m \operatorname{then}(\operatorname{ret} \operatorname{zero}) \operatorname{else}\{m^*; \operatorname{do} r\}))
```

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

```
\operatorname{proc}\left(x:\tau\right)\left\{m\right\}\stackrel{\text{\tiny def}}{=}\lambda x:\tau.\operatorname{cmd}(m) \qquad \qquad \operatorname{call} e_{1}(e_{2})\stackrel{\text{\tiny def}}{=}\operatorname{do}e_{1}(e_{2})
```

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

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