PFPL Supplement: Derived Forms for Modernized Algol*

Robert Harper

Fall, 2020

1 Introduction

The imperative language MA is formulated to expose its semantic structure, rather than to facilitate programming. It is helpful to define derived forms that are reminiscent of familiar imperative languages, such as the C language and its derivatives. There is, of course, no end to the possible derived forms corresponding to imperative language constructs.

2 Derived Forms

Blocks Blocks of commands are defined as follows:

$$\begin{aligned}
\text{RET}\, e &\triangleq \texttt{ret}(\, e \,) \\
\{x \leftarrow m_1 \,;\, m_2\} &\triangleq \texttt{bnd}(\, \texttt{cmd}(\, m_1 \,) \,;\, x \,.\, m_2 \,) \\
\{x_1 \leftarrow m_1 \,;\, \ldots \,;\, x_k \leftarrow m_k \,;\, \text{RET}\, e\} &\triangleq \{x_1 \leftarrow m_1 \,;\, \ldots \{x_k \leftarrow m_k \,;\, \text{RET}\, e\}\} \\
\text{DO}\, m &\triangleq \{x \leftarrow m \,;\, \text{RET}\, x\} \\
\text{RET} &\triangleq \texttt{ret}(\, \langle\rangle \,) \\
\{m_1 \,;\, m_2\} &\triangleq \{_ \leftarrow \texttt{cmd}(\, m_1 \,) \,;\, m_2\} \\
\{m_1 \,;\, \ldots \,;\, m_k\} &\triangleq \{_ \leftarrow m_1 \,;\, \ldots _ \leftarrow m_k \,;\, \text{RET}\, \langle\rangle\}
\end{aligned}$$

It is common to intermix named with unnamed bindings in a block, with the understanding that the unnamed forms, when intermixed with the named forms, are to be regarded as binding an anonymous variable.

Assignables Multiple declarations are defined as follows:

$$\text{DCL}\, a_1 := e_1 \ldots a_k := e_k \,\text{IN}\, m \triangleq \texttt{dcl}(\, e_1 \,;\, a_1 \,.\, \ldots \texttt{dcl}(\, e_k \,;\, a_k \,.\, m \,)\,)$$

Commands for getting and setting assignables are defined as follows:

$$\begin{aligned}
\textsc{get}\, a &\triangleq \mathtt{get}[\, a\,] \\
\textsc{set}\, a \mathtt{:=} e &\triangleq \mathtt{set}[\, a\,](\, e\,) \\
\mathtt{++}a &\triangleq \{x \leftarrow \textsc{get}\, a\,;\, \textsc{set}\, a \mathtt{:=} x+1\} \\
\mathtt{--}a &\triangleq \{x \leftarrow \textsc{get}\, a\,;\, \textsc{set}\, a \mathtt{:=} x-1\} \\
a\mathtt{++} &\triangleq \{x \leftarrow \textsc{get}\, a\,;\, \{\textsc{set}\, a \mathtt{:=} x+1\,;\, \textsc{ret}\, x\}\} \\
a\mathtt{--} &\triangleq \{x \leftarrow \textsc{get}\, a\,;\, \{\textsc{set}\, a \mathtt{:=} x-1\,;\, \textsc{ret}\, x\}\}
\end{aligned}$$

Conditional Branch The conditional command, which branches on whether an assignable is zero, is defined as follows:

$$\begin{aligned}
\textsc{ifz}\, m\, \textsc{then}\, m_0\, \textsc{else}\, x \rightarrow m_1 &\triangleq \{y \leftarrow m\,;\, \textsc{do}\,(\,\mathtt{ifz}(\, y\,;\, \mathtt{cmd}(\, m_1\,)\,;\, x\,.\,\mathtt{cmd}(\, m_2\,)\,)\,)\} \\
\textsc{ifz}\, m\, \textsc{then}\, m_0\, \textsc{else}\, m_1 &\triangleq \textsc{ifz}\, m\, \textsc{then}\, m_0\, \textsc{else}\, _ \rightarrow m_1
\end{aligned}$$

Notice that the conditional expression returns an encapsulated command, which must be activated by the DO. A conditional based on a boolean test would be easily definable along similar lines.

Procedures (Recursive) procedures and procedure calls are defined as follows:

$$\begin{aligned}
\tau_1 \Rightarrow \tau_2 &\triangleq \tau_1 \rightharpoonup \mathtt{cmd}(\, \tau_2\,) \\
\textsc{proc}\, p(\, x : \tau_1\,) \mathbin{\dot{:}} \tau_2\, \textsc{is}\, m &\triangleq \mathtt{fun}\{\tau_1\,;\, \mathtt{cmd}(\, \tau_2\,)\}(\, p\,.\,x\,.\,\mathtt{cmd}(\, m\,)\,) \\
\textsc{call}\, e\,(\, e_1\,) &\triangleq \textsc{do}\,(\,\mathtt{ap}(\, e\,;\, e_1\,)\,) \\
\textsc{call}\, e &\triangleq \textsc{call}\, e\,(\, \langle\rangle\,)
\end{aligned}$$

Higher-order recursive procedures—with a call-by-name semantics—were introduced in Algol 60! To preserve the stack discipline of storage management, procedures may take procedures (or encapsulated commands) as arguments, but they may not return either of them as result. (This is expressed by the mobility restriction in **MA**.)

Loops The while command, governed by whether an assignable is zero, is defined as follows:

$$\begin{aligned}
\textsc{while}\, m_1\, \textsc{do}\, x \rightarrow m_2 &\triangleq \textsc{call}\, w, \;\; \textit{where} \\
w &\triangleq \textsc{proc}\, w(\, _ : \mathtt{1}\,) \mathbin{\dot{:}} \mathtt{1}\, \textsc{is}\, \textsc{ifz}\, m_1\, \textsc{then}\, \textsc{ret}\, \textsc{else}\, x \rightarrow \{m_2\,;\, \textsc{call}\, w\} \\
\textsc{while}\, m_1\, \textsc{do}\, m_2 &\triangleq \textsc{while}\, m_1\, \textsc{do}\, _ \rightarrow m_2
\end{aligned}$$

There is no end to the varieties of looping constructs that are also definable in **MA**.

3 Examples

An imperative version of the good old factorial function may be defined by the following procedure:

$$\begin{aligned}
\mathit{fact} &\triangleq \textsc{proc}\, \mathit{fact}(\, n : \mathtt{nat}\,) \mathbin{\dot{:}} \mathtt{nat}\, \textsc{is}\, \textsc{dcl}\, \mathit{ans} := \overline{1}\,;\, i := n\, \textsc{in}\, \mathit{body} \\
\mathit{body} &\triangleq \{\textsc{while}\, \textsc{get}\, i\, \textsc{do}\, n \rightarrow \{\textsc{set}\, \mathit{ans} \times\!= n\,;\, i\mathtt{--}\}\,;\, \{a \leftarrow \textsc{get}\, \mathit{ans}\,;\, \textsc{ret}\, a\}\}
\end{aligned}$$

wherein we have used the "arithmetic assignment" command

$$\text{SET}\, a \times\!= e \triangleq \{x \leftarrow \text{GET}\, a\, ;\, \text{SET}\, a\, \texttt{:=}\, e \times x\},$$

which multiplies the contents of an assignable a by an amount given by e.

References may be defined as capabilities to get and set an assignable.

$$\begin{aligned}
\tau\ \text{REF} &\triangleq (\ \tau\ \texttt{cmd}\ ) \times (\ \tau \Rightarrow \tau\ ) \\
\text{NEWREF} &\triangleq \text{PROC}\ _(\ x : \tau\ ) \div \tau\ \text{REF IS DCL}\ a := x\ \text{IN RET}\ \langle \mathit{get}, \mathit{set} \rangle,\ \ \mathit{where} \\
\mathit{get} &\triangleq \texttt{cmd}(\ \text{GET}\, a\ ) \\
\mathit{set} &\triangleq \text{PROC}\ _(\ x : \tau\ ) \div \tau\ \text{IS SET}\, a\ \texttt{:=}\ x \\
\text{GETREF} &\triangleq \text{PROC}\ _(\ r : \tau\ \text{REF}\ ) \div \tau\ \text{IS DO}\ r \cdot \texttt{l} \\
\text{SETREF} &\triangleq \text{PROC}\ _(\ \langle r, x \rangle : \tau\ \text{REF} \times \tau\ ) \div \tau\ \text{IS DO CALL}\ r \cdot \texttt{r}\ (\ x\ )
\end{aligned}$$

Capabilities, like references, are fully compatible with the stack discipline, because the defined type $\tau\ \text{REF}$ is not mobile, because no procedure types are mobile. Thus, a capablity is confined to the scope of the assignable to which it provides access; it can neither be returned from that scope, nor stored into another assignable.

References

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, Cambridge, England, Second edition, 2016.