The Von Neumann machine as a functional program

0. A simple case

Our starting point is the following tail-recursive definition of a binary operator $\otimes_0$ in terms of a given binary operator $\oplus$:

$$(0a) \quad s \otimes_0 [\,] = s$$
$$(0b) \quad s \otimes_0 ([b] +\!\!+ x) = (s \oplus b) \otimes_0 x$$

In this definition s has type S, b has type B, $\oplus$ has type $S \times B \to S$, and x has type $[B]$ ("list of B"); hence $\otimes_0$ has type $S \times [B] \to S$.

Operator $\otimes_0$ can be considered as a programmable machine. In such a view, parameter s represents the <u>state</u> of the machine and x can be interpreted as the list of instructions to be executed —the <u>program</u>—; $s \oplus b$, then, is the (new) state of the machine as a result of executing instruction b in state s. The case $s \otimes_0 [\,] = s$ represents termination of the machine's activity because it has reached the end of the program.

We now study various transformations of the above definition into equivalent definitions, albeit that in the new definitions list x is represented differently. The purpose of these transformations is to obtain versions that more accurately describe (the operation of) computers of the Von Neumann type. Thus, we intend to show that definitions like (0) can be implemented as machine code programs in a straightforward way, particularly so when

the elements of $\mathcal{B}$ correspond directly to instructions in the repertoire of the machine.

example: With $S = [Int]$ and $\mathcal{B} = Int \cup Ops$, where Ops is some (finite) subset of $Int \times Int \to Int$, we can define $\otimes_0$ by, with $m, n \in Int$ and $\oplus \in Ops$:

$$\begin{aligned} s \otimes_0 [] &= s \\ s \otimes_0 ([n] +\!\!+ x) &= (s +\!\!+ [n]) \otimes_0 x \\ (s +\!\!+ [m,n]) \otimes_0 ([\oplus] +\!\!+ x) &= (s +\!\!+ [m \oplus n]) \otimes_0 x \,. \end{aligned}$$

$\otimes_0$ is an evaluator for simple integer expressions in postfix form; in this definition, an integer n (in list x) represents an instruction to "place integer value n on top of the stack s" and an operator $\oplus$ represents an instruction to "replace the topmost two stack elements m and n by $m \oplus n$".

□

First, we assume $!$ to be a value satisfying $! \notin \mathcal{B}$ and we extend the definition of $\otimes_0$ into the definition of a new operator $\otimes_1$, as follows:

$$\begin{aligned} &(1a) \quad s \otimes_1 [] = s \\ &(1b) \quad s \otimes_1 ([b] +\!\!+ x) = (s \oplus b) \otimes_1 x \\ &(1c) \quad s \otimes_1 ([!] +\!\!+ x) = s \end{aligned}$$

Now we have $s \otimes_1 (x +\!\!+ [!] +\!\!+ y) = s \otimes_0 x$ for all x not containing $!$; this is, of course, proved by straightforward mathematical induction on the length of x. The value $!$ can be interpreted as a "halt" instruction explicitly marking the end of the program. Rule (1a) in

this definition is superfluous provided all lists x in $s \otimes_1 x$ contain at least one $!$: in that case rule (1c) replaces rule (1a). In what follows we shall tacidly assume $! \in x$ for all x in $s \otimes_1 x$ under consideration.

Second, we observe that all lists x occurring in the evaluation of $s \otimes_1 X$, for some fixed X, satisfy $x = X{\downarrow}p$ *) for some natural p. Therefore, when we are only interested in $s \otimes_1 X$ we can represent x by a single natural parameter p, as follows:

$$
\begin{array}{lll}
(2) \quad s \otimes_2 p = & \underline{\text{if}} \; X.p \in B \to & (s \oplus b) \otimes_2 (p+1) \quad [\![\, b = X.p \,]\!] \\
& [\!] \; X.p = ! \to & s \\
& \underline{\text{fi}} &
\end{array}
$$

Thus defined, $\otimes_2$ satisfies $s \otimes_2 p = s \otimes_1 (X{\downarrow}p)$, for all $p : 0 \leq p < \#X$ (provided $! \in X{\downarrow}p$); in particular we have $s \otimes_2 0 = s \otimes_1 X$. In this way, the part of the program still to be executed is represented by a single natural number, the parameter p; traditionally, p is called the instruction pointer, program index, or program counter [sic].

1. Subroutines

We now take a somewhat more complicated version of (0) as our starting point:

$$
\begin{array}{lll}
(3a) & s \otimes_3 [\,] & = s \\
(3b) & s \otimes_3 ([b] +\!\!+ x) & = (s \oplus b) \otimes_3 x \\
(3c) & s \otimes_3 ([c] +\!\!+ x) & = s \otimes_3 (s \odot c +\!\!+ x) \,,
\end{array}
$$

---

*) Actually, it is sufficient that $s \otimes_1 x = s \otimes_1 (X{\downarrow}p)$, which is weaker.

where c has type C, for some type C with $B \cap C = \emptyset$, and where $\odot$ has type $S \times C \to [B \cup C]$. (Notice that x now has type $[B \cup C]$.) Rule (3c) can be made more general in that s in its right-hand side may be replaced by any expression depending on s and c without any problem, but for the sake of simplicity we shall not do so. Moreover, we shall not concern ourselves here with well-definedness of $\otimes_3$: we just <u>assume</u> that the recursion in (3b) and (3c) is well-founded.

In order to avoid the catenation operator $+\!\!+$ in (3c)'s right-hand side we now represent x by a list of lists, namely by $x = \text{flatten}.xs$ for some xs of type $[[B \cup C]]$. That is, we introduce a new operator $\otimes_4$ with specification:

$$s \otimes_4 xs \quad = \quad s \otimes_3 \text{flatten}.xs \; .$$

Conversely, we have $s \otimes_3 x = s \otimes_4 [x]$. A tail-recursive definition of $\otimes_4$ can be derived easily from this specification and from (3), yielding:

$$\begin{aligned}
&(4a) && s \otimes_4 [\,] && = && s \\
&(4b) && s \otimes_4 ([[\,]] +\!\!+ xs) && = && s \otimes_4 xs \\
&(4c) && s \otimes_4 ([[b] +\!\!+ x] +\!\!+ xs) && = && (s \oplus b) \otimes_4 ([x] +\!\!+ xs) \\
&(4d) && s \otimes_4 ([[c] +\!\!+ x] +\!\!+ xs) && = && s \otimes_4 ([s \odot c] +\!\!+ [x] +\!\!+ xs)
\end{aligned}$$

(The for this definition relevant properties of flatten are assumed to be known.)

Rule (4a) may be replaced safely by:

$$(4a') \quad s \otimes_4 [[\,]] \quad = \quad s \; ,$$

provided that all applications $s \otimes_4 xs$ satisfy $xs \neq []$, as is the case with $s \otimes_4 [x]$.

We now apply the same kind of transformations as we did in the previous section. First, we introduce the value ! again, now satisfying $! \notin B \cup C$, as an explicit end-of-list marker. We recall from the previous section that $x +\!\!+ [!] +\!\!+ y$ and $x +\!\!+ [!] +\!\!+ z$ are <u>equivalent</u>, for all x, y, z. Incorporating this in our definitions we obtain:

$$s \otimes_4 xs \quad = \quad s \otimes_5 xs' \text{ , for nonempty } xs \text{ ,}$$

where xs' is obtained from xs by suffixing each list in xs with a ! (That is, $xs' = (+\!\!+[!]) \mathbin{@} xs$) The following definition of $\otimes_5$ does the job:

$$\begin{array}{lll}
(5a) & s \otimes_5 ([[!] +\!\!+ x] +\!\!+ xs) = s & , \; xs = [] \\
(5b) & s \otimes_5 ([[!] +\!\!+ x] +\!\!+ xs) = s \otimes_5 xs & , \; xs \neq [] \\
(5c) & s \otimes_5 ([[b] +\!\!+ x] +\!\!+ xs) = (s \oplus b) \otimes_5 ([x] +\!\!+ xs) & \\
(5d) & s \otimes_5 ([[c] +\!\!+ x] +\!\!+ xs) = s \otimes_5 ([s \circ c +\!\!+ [!]] +\!\!+ [x] +\!\!+ xs) &
\end{array}$$

<u>Aside:</u> a more "readable" version of this definition is obtained by use of the "cons"-operator ;, with $[x] +\!\!+ xs = x ; xs$, and by exploitation of xs's being nonempty: its first element can be represented as an additional parameter. If we call the new function F, its specification is:

$$F.s.x.xs \quad = \quad s \otimes_5 (x ; xs) \text{ , and its definition:}$$

$$\begin{array}{lll}
F.s.(!;x).[] & = & s \\
F.s.(!;x).(y;xs) & = & F.s.y.xs \\
F.s.(b;x).xs & = & F.(s \oplus b).x.xs \\
F.s.(c;x).xs & = & F.s.(s \circ c +\!\!+ [!]).(x;xs)
\end{array}$$

The technique of representing the head of a list by an additional parameter can be considered as the functional counterpart of keeping the top element of a stack in a processor register.

□

Second, we assume the availability of a list X with the property that every "relevant" list is equivalent to $X\downarrow p$, for some natural p. In particular we assume that $s \odot c +\!\!+ [!]$ is equivalent to some $X\downarrow p$, and we shall denote this p by $s\bar{\odot}c$; so, we have:

$$s \odot c +\!\!+ [!] \text{ is equivalent to } X\downarrow(s\bar{\odot}c) .$$

All lists involved are now represented by natural numbers. This change of representation is reflected in the specification and definition of a new operator $\otimes_6$, as follows:

$$s \otimes_6 ps = s \otimes_5 ((X\downarrow) @ ps) \text{ , where}$$

$$\begin{aligned}(6)\quad s \otimes_6 ([p] +\!\!+ ps) = \ & \textbf{if } X.p = ! \wedge ps = [] \rightarrow s \\ & [] \ X.p = ! \wedge ps \neq [] \rightarrow s \otimes_6 ps \\ & [] \ X.p \in B \rightarrow (s \oplus b) \otimes_6 ([p+1] +\!\!+ ps) \\ & \qquad \| [\ b = X.p \ ] \| \\ & [] \ X.p \in C \rightarrow s \otimes_6 ([s\bar{\odot}c] +\!\!+ [p+1] +\!\!+ ps) \\ & \qquad \| [\ c = X.p \ ] \| \\ & \textbf{fi}\end{aligned}$$

In this definition list ps can be interpreted as a list of instruction pointers, also called the "stack of return addresses"; instructions of type C correspond in this view with "subroutine calls" whereas $!$ now represents a "subroutine return"

2. Jumps

If rule (3c) would have the shape:

$$s \otimes_3 ([c] +\!\!+ x) \quad = \quad s \otimes_3 (s \circ c) \,,$$

the corresponding line in definition (6) would look like:

$$[]\ X.p \in C \rightarrow s \otimes_6 ([s \bar{\circ} c] +\!\!+ ps)\ \ |[\ c = X.p\ ]|\ .$$

Now, the instructions in C are the functional counterparts of (either conditional or unconditional) "jumps".

Eindhoven, 20 september 1994

Rob R. Hoogerwoord