

April 17, 1984

The "Program" Approach to FP Programming.

John Backus

1) Proposal: A program is a function p' that maps the domain consisting of pairs $\langle \text{object}, \text{state} \rangle$ into itself. This state contains:

- a) a file system, fs .
- b) a set of input streams, one for each input device.
- c) a set of output streams, one for each output device.
- d) one or more "screens", together with cursor and window data; these represent the current condition of the display and displayable data. The screen interface can be controlled either by its output stream or by accessing the screen data, changing it and then replacing it in the state.

There will be a set of primitive programs that can change the state. Examples:

$\text{Read}': \langle x, s \rangle = \langle \langle x, y \rangle, s' \rangle$ where y is the next input and s' is s with the input deleted from the head of the input stream.

$\text{Write}': \langle \langle x, y \rangle, s \rangle = \langle \langle x, y \rangle, s' \rangle$ where s' has x at the head of its output stream [depending which way output streams are arranged].

$\text{Get}': \langle x, s \rangle = \langle \langle x, fs \rangle, s \rangle$ where fs is the file system in s .

$\text{Put}': \langle \langle x, y \rangle, s \rangle = \langle \langle x, y \rangle, s' \rangle$ where x becomes the file system in s' .

Getscr' and Putscr' would act similarly, to get and replace the screen data in the state.

Only such primitive programs can alter and/or access the state. Programs that neither access nor change the state are called functions; if f' is such a program [a very pure program in EW's terms], then

$f': \langle x, s \rangle = \langle y, s \rangle$ where for some f , $f: x = y$.

we associate the function f with the program f' , and think of the mapping

$$f:x = y$$

as if this were the action of the program f' . In fact, it may be useful, sometimes, to think of programs as quasi-functions on objects [they may depend on the state as well as the object], quasi-functions that also have "side effects" [they may change the state], thus one could think of Write as mapping $\langle x, y \rangle$ into itself, with the side effect of outputting x .

Note that programs that are not functions always preserve a part of their object-argument. This is intended to avoid the need to form constructions of programs other than functions. Hopefully, we can entirely do without constructions of non-function programs; currently it is proposed that the construction of programs, if one is a non-function, is meaningless.

Perhaps the best way to illustrate this approach is to give a program, with comments, for writing checks. Since the principal way of putting programs together is composition, I will write them from top to bottom instead of right to left.

At the top level let us assume the existence of an interactive program 'wrtcks' that transforms: $\langle \text{accts}, \text{ckreg} \rangle \rightarrow \text{ckreg}$, where *accts* is the file of accounts and *ckreg* is the file of check register entries. Using *wrtcks*, the program *ckwrt* gets *accts* and *ckreg* from *fs*, and forms the pair $\langle \text{accts}, \text{ckreg} \rangle$ as the argument for *wrtcks*. After *wrtcks* produces *ckreg'* as its result [accompanied by the interaction with the user, and with the side effect of printing checks], the program ends by getting *fs* again, replacing *ckreg* by *ckreg'* in it, and then putting this altered *fs* back into the state.

<u>Program</u>	<u>Result [object]</u>	<u>New state / comment</u>
<u>Def ckwrt</u>		
	x	s
get	<x,fs>	
2	fs	
[^accts, ^ckreg]	<accts, ckreg>	^accts = fetch accts
wrtcks	ckreg'	s' = s with i/o streams changed by interaction
get	<ckreg',fs>	
'ckreg.+id	<ckreg,ckreg',fs>	
store	fs'	fs' contains obj ckreg' in cell named ckreg
put	fs'	s'' = s' with fs replaced by fs
<u>Enddef ckwrt</u>		

Or, in FP style:

Def ckwrt = put@('ckreg.+id)@get@wrtcks@[^accts,^ckreg]@2@get

In defining wrtcks we shall use a program, **display**, that interactively transforms <message,file,x> into <file_item,x>. It displays the message and the file, allows one to pageup and pagedown in the file, move the cursor, and select a file item by pointing at it and pressing enter; this file item is the result of display.

We also use the program called **input**, which takes <x,y> and produces <z,y> by outputting x and then reading z as the response to x.

Def wrtcks

<accts, ckreg>

```

['select_acct, 1, id]      <select_acct, accts, <accts,ckreg>>
display                    <acct, <accts,ckreg>>
[['enter_amt,1],a1]       <<enter_amt,acct>,<acct,accts,ckreg>>
input                      <amt, <acct,accts,ckreg>>
        outputs "enter amt", acct ; gets amt in response
[['ok?,1@2,1],1.+2]       <<'ok?,acct,amt>,<amt,acct,accts,ckreg>>
input                      <resp,<amt,acct,accts,ckreg>>
l='yes -> [2,1]@2; wrtchks@[3,4]@2
                                <<acct,amt>,<accts,ckreg>> if resp=yes
                                wrtchks:<accts,ckreg>          ow
print                      <<acct,amt>,<accts,ckreg>>          print check
[1,[1@2,1.+2@2]           <<acct,amt>,<accts,(<acct,amt>.+ckreg)>>
                                update ckreg
input@['quit?,2]          <resp,<accts,ckreg>>          print "quit?"
l='yes -> 2@2; wrtchks@2

```

Enddef wrtchks

```

Def wrtchks = (l='yes -> 2@2; wrtchks@2) @ input @ ['quit?,2] @
                [1,[1@2,1.+2@2] @ print @ (l='yes -> [2,1]@2; wrtchks@[3,4]@2) @
                [['ok?,1@2,1],1.+2] @ input @ [['enter_amt,1],a1] @ display @
                ['select_acct,1,id]

```

These programs are certainly not elegant, but perhaps by examining their properties we can do better. I hope we can find combining forms that will make it unnecessary to always write such programs recursively. In any case I would like to understand what can be done or not done to optimize them. It seems that for very interactive programs, which are larded with i/o operations, that it may just be intrinsically not possible to rearrange them very much. On the other hand, it seems that programs that are mostly "functional" can be optimized by FP algebraic methods.

Goals

- 1) Establish a system in which it is always clear what the state of the

system is. That is, it should always be clear to the user what the current status of all the input and output devices is, and what the contents of the permanent store or file system are. Other approaches, in which there is no distinguished state, seem to have a problem in this respect. I believe the program approach makes it clear what the state is under all circumstances.

2) Establish a useful notion of the "purpose" of a program as a transformation on objects [even though such transformations may not be proper functions]. Thus one may consider that the purpose of an editor, or a formatter, is to transform documents. If one does, then the composition of the purposes of such programs are meaningful and may be achieved by the composition of the programs.

If, on the other hand, one considers such interactive programs to be strict functions from $\langle \text{document}, \text{instream} \rangle$ to $\langle \text{document}, \text{outstream} \rangle$ [or something similar], then one cannot usefully regard this mapping as their "purpose", for if one does then their purposes [and the programs themselves] have no meaningful composition.

3) Find ways to separate the clean [functional] parts of programs from the dirty [non-function] parts. Classify programs into perhaps the following classes:

- a) functions do not depend on the state nor change it
- b) pure programs do not change the state
- c) interactive programs do not change the file system in the state
- d) programs

Perhaps these or other classes of programs will have, on a decreasing scale, different sets of algebraic properties that can be used in transforming them. I hope we can get a clearer picture of what transformations can be meaningfully made on programs of these different classes [and what ones are not realistically meaningful¹⁾].