

position is that it permits the definition of new functional forms, in effect, merely by defining new functions. It also permits one to write recursive functions without a definition.

We give one more example of a controlling function for a functional form: Def pCONS := otapplyotlodistr. This definition results in <CONS,fi fn>--where the f_i are objects--representing the same function as [pfl pfn]. The following shows this.

$$\begin{aligned} & (p<CONS,fi fn>):X \\ &= (\#CONS):<<CONS, fi fn >,X> \\ &\text{by metacomposition} \\ &= aapplyotlodistr:<<CONS,fi fn>,X> \\ &\text{by def of pCONS} \\ &= aapply:<<f_i,x> <f_n,X>> \\ &\text{by def of tl and distr and o} \\ &= <apply:<f_i,x> apply:<f_n, X>> \\ &\text{by def of a} \\ &= <(f_i:x) (f_n:X)> \\ &\text{by def of apply.} \end{aligned}$$

In evaluating the last expression, the meaning function will produce the meaning of each application, giving $p_j \sim :x$ as the i th element.

Usually, in describing the function represented by a sequence, we shall give its overall effect rather than show how its controlling operator achieves that effect. Thus

we would simply write

$$(p<\text{CONS}, f_1 \dots f_n>):x = <(f_1:x) \dots (f_n:x)>$$

instead of the more detailed account above.

We need a controlling operator, COMP, to give us sequences representing the functional form composition.

We take pCOMP to be a primitive function such that,

for all objects x ,

$$(p<\text{COMe}, f_1 \dots f_n>):x$$
$$= (f_1:(f_2:(\dots (f_n:x)\dots))) \text{ for } n \geq 1.$$

(I am indebted to Paul Me Jones for his observation that ordinary composition could be achieved by this primitive function rather than by using two composition rules in the basic semantics, as was done in an earlier paper [2].)

Although FFP systems permit the definition and investigation of new functional forms, it is to be expected that most programming would use a fixed set of forms (whose controlling operators are primitives), as in FP, so that the algebraic laws for those forms could be employed, and so that a structured programming style could be used based on those forms.

In addition to its use in defining functional forms, metacomposition can be used to create recursive functions directly without the use of recursive definitions of the form $\text{Def } f \sim E(f)$. For example, if pMLAST

nullotlo2 ~

$\text{lo2}; \text{applyo}[1, \text{tlo2}], \text{ then } p\langle \text{MLAST} \rangle$

$\text{-} =$

last, where $\text{last}: x \mapsto x = \langle x_1 \dots x_n \rangle \sim X \sim$; &. Thus the

operator $\langle \text{MLAST} \rangle$

works as follows:

$\#(\langle \text{MLAST} \rangle : \langle A, B \rangle)$

633

$= \#(p\text{MLAST} : \langle \langle \text{MLAST} \rangle, \langle A, B \rangle \rangle)$

by metacomposition

$= \#(\text{applyo}[1, \text{tlo2}] : \langle \langle \text{MLAST} \rangle, \langle A, B \rangle \rangle)$

$= \sim t(\text{apply} : \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle)$

$= \#(\langle \text{MLAST} \rangle : \langle B \rangle)$

$= ix(p\text{MLAST} : \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle)$

$= \#(\text{lo2} : \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle)$

$= B.$

13.3.3 Summary of the properties of p and $\#$. So far

we have shown how p maps atoms and sequences into

functions and how those functions map objects into

expressions. Actually, p and all FFP functions can be

extended so that they are defined for all expressions.

With such extensions the properties of p and \sim can be

summarized as follows:

1) $\# E$ [expressions \rightarrow objects].

2) If x is an object, $\#x = x$.

3) If e is an expression and $e = \langle e_1 \dots e_n \rangle$, then

$\#e = \langle \#e_1, \dots, \#e_n \rangle$.

4) $p \in [\text{expressions} \sim [\text{expressions} \sim \text{expressions}]]$.

5) For any expression e , $p_e = p_{\sim e}$.

6) If x is an object and e an expression, then

$ox:e = px:(ge)$.

7) If x and y are objects, then $\#(x:y) = \#(Ox:y)$. In

words: the meaning of an FFP application $(x:y)$ is found

by applying px , the function represented by x , to y and

then finding the meaning of the resulting expression

(which is usually an object and is then its own meaning).

13.3.4 Cells, fetching, and storing. For a number of reasons it is convenient to create functions which serve as names. In particular, we shall need this facility in describing the semantics of definitions in FFP systems.

To introduce naming functions, that is, the ability to fetch the contents of a cell with a given name from a store (a sequence of cells) and to store a cell with given name and contents in such a sequence, we introduce objects called cells and two new functional forms, fetch and store.

Cells

A cell is a triple $\langle \text{CELL}, \text{name}, \text{contents} \rangle$. We use this form instead of the pair $\langle \text{name}, \text{contents} \rangle$ so that cells can be distinguished from ordinary pairs.

Fetch

The functional form fetch takes an object n as its parameter (n is customarily an atom serving as a name);

it is written $l'n$ (read "fetch n "). Its definition for objects

n and x is

$l'n:x \rightarrow x = \sim \sim \#$; $\text{atom}:x \sim \pm$;

$(l:x) = \langle \text{CELL}, n, c \rangle \sim c$; $\sim \text{'not}l:x$,

where $\#$ is the atom "default." Thus $l'n$ (fetch n) applied

to a sequence gives the contents of the first cell in the

sequence whose name is n ; If there is no cell named n ,

the result is default, $\#$. Thus $l'n$ is the name function for

the name n . (We assume that $p\text{FETCH}$ is the primitive

function such that $p\langle \text{FETCH}, n \rangle \sim l'n$. Note that $\sim n$

simply passes over elements in its operand that are not

cells.)

Communications

August 1978

of

Volume 21

the ACM

Number 8

position is that it permits the definition of new functional forms, in effect, merely by defining new functions. It also permits one to write recursive functions without a definition.

We give one more example of a controlling function for a functional form: **Def** $\rho\text{CONS} \equiv \alpha\text{apply} \circ \text{tl} \circ \text{distr}$. This definition results in $\langle \text{CONS}, f_1, \dots, f_n \rangle$ —where the f_i are objects—representing the same function as $[\rho f_1, \dots, \rho f_n]$. The following shows this.

$(\rho \langle \text{CONS}, f_1, \dots, f_n \rangle):x$
 $= (\rho\text{CONS}): \langle \langle \text{CONS}, f_1, \dots, f_n \rangle, x \rangle$

by metacomposition

$= \alpha\text{apply} \circ \text{tl} \circ \text{distr}: \langle \langle \text{CONS}, f_1, \dots, f_n \rangle, x \rangle$

by def of ρCONS

$= \mu(\rho\text{MLAST}: \langle \langle \text{MLAST} \rangle$

$= \mu(\text{apply} \circ [1, \text{tl} \circ 2]: \langle \langle \text{MLAST} \rangle,$

$= \mu(\text{apply}: \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle)$

$= \mu(\langle \text{MLAST} \rangle: \langle B \rangle)$

$= \mu(\rho\text{MLAST}: \langle \langle \text{MLAST} \rangle, \langle B \rangle$

$= \mu(1 \circ 2: \langle \langle \text{MLAST} \rangle, \langle B \rangle \rangle)$

$= B$.

13.3.3 Summary of the properties

we have shown how ρ maps atoms functions and how those functions expressions. Actually, ρ and all FF extended so that they are defined With such extensions the properties summarized as follows:

1) $\mu \in [\text{expressions} \rightarrow \text{objects}]$