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** $\rho CONS \equiv \alpha\text{apply}\circ\text{tl}\circ\text{distr}$. This definition results in $<CONS,f_1, \dots ,f_n>$—where the f_i are objects—representing the same function as $[\rho f_1, \dots , \rho f_n]$. The following shows this.

$$\begin{aligned}(\rho<CONS,f_1, \dots ,f_n>):x &= (\rho CONS):<<CONS,f_1, \dots ,f_n >,x> \quad \text{by metacomposition}\\ &= \alpha\text{apply}\circ\text{tl}\circ\text{distr}:<<CONS,f_1, \dots ,f_n>,x> \quad \text{by def of } \rho CONS\\ &= \alpha\text{apply}:<<f_1,x>, \dots , <f_n,x>> \quad \text{by def of tl and distr and } \circ\\ &= <\text{apply}:<f_1,x>, \dots , \text{apply}:<f_n,x>> \quad \text{by def of } \alpha\\ &= <(f_1:x), \dots , (f_n:x)> \quad \text{by def of apply.}\end{aligned}$$

In evaluating the last expression, the meaning function μ will produce the meaning of each application, giving $\rho f_i:x$ as the ith 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

$$(\rho<CONS,f_1, \dots ,f_n>):x = <(f_1:x), \dots , (f_n:x)>$$

instead of the more detailed account above.

We need a controlling operator, *COMP*, to give us sequences representing the functional form composition. We take $\rho COMP$ to be a primitive function such that, for all objects x,

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

(I am indebted to Paul McJones 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 **Def** $f \equiv E(f)$. For example, if $\rho MLAST \equiv \text{null}\circ\text{tl}\circ 2 \rightarrow 1\circ 2; \text{apply}\circ[1, \text{tl}\circ 2]$, then $\rho<MLAST> \equiv \text{last}$, where $\text{last}:x \equiv x = <x_1, \dots, x_n> \rightarrow x_n; \bot$. Thus the operator $<MLAST>$ works as follows:

$$\begin{aligned}&\mu(<MLAST>:<A,B>)\\ &= \mu(\rho MLAST:<<MLAST>,<A,B>>) \quad \text{by metacomposition}\\ &= \mu(\text{apply}\circ[1, \text{tl}\circ 2]:<<MLAST>,<A,B>>)\\ &= \mu(\text{apply}:<<MLAST>,<B>>)\\ &= \mu(<MLAST>:<B>)\\ &= \mu(\rho MLAST:<<MLAST>,<B>>)\\ &= \mu(1\circ 2:<<MLAST>,<B>>)\\ &= B.\end{aligned}$$

13.3.3 Summary of the properties of ρ and μ. So far we have shown how ρ maps atoms and sequences into functions and how those functions map objects into expressions. Actually, ρ and all FFP functions can be extended so that they are defined for all expressions. With such extensions the properties of ρ and μ can be summarized as follows:

1) $\mu \in$ [expressions $\rightarrow$ objects].
2) If x is an object, $\mu x = x$.
3) If e is an expression and $e = <e_1, \dots , e_n>$, then $\mu e = <\mu e_1, \dots , \mu e_n>$.
4) $\rho \in$ [expressions $\rightarrow$ [expressions $\rightarrow$ expressions]].
5) For any expression e, $\rho e = \rho(\mu e)$.
6) If x is an object and e an expression, then $\rho x:e = \rho x:(\mu e)$.
7) If x and y are objects, then $\mu(x:y) = \mu(\rho x:y)$. In words: the meaning of an FFP application $(x:y)$ is found by applying ρx, 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 $<CELL,name,contents>$. We use this form instead of the pair $<name,contents>$ 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 $\uparrow n$ (read "fetch n"). Its definition for objects n and x is

$$\uparrow n:x \equiv x = \phi \rightarrow \#; \text{atom}:x \rightarrow \bot; (1:x) = <CELL,n,c> \rightarrow c; \uparrow n\circ\text{tl}:x,$$

where # is the atom "default." Thus $\uparrow n$ (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 $\uparrow n$ is the name function for the name n. (We assume that $\rho FETCH$ is the primitive function such that $\rho<FETCH,n> \equiv \uparrow n$. Note that $\uparrow n$ simply passes over elements in its operand that are not cells.)