APPLICATIVE MULTIPROGRAMMING*

Daniel P. Friedman
David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 72
APPLICATIVE MULTIPROGRAMMING

DANIEL P. FRIEDMAN
DAVID S. WISE

REVISED: APRIL, 1979

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 A01, MCS75-08145, and MCS77-22325.

Applicative Multiprogramming*

Daniel P. Friedman
David S. Wise
Computer Science Department
Indiana University
Bloomington, Indiana 47405

Abstract. This paper defines and describes a new data structure construction mechanism which enables applicative programming techniques to handle problems for which computational effort must be distributed across several candidate expressions. This technique may be used on purely internal structures or may be used to organize computation and communication with external asynchronous events. The new constructor, _frons_, takes its place alongside the classic sequence constructor, _cons_, extending the user's data structure from the list to the newly introduced fern. A fern is a natural generalization both of a sequence and of a _multiset_ (an unordered set allowing duplicates). Content must be specified as a fern is built, but order may remain unspecified until elements are accessed; then the elements specified by convergent expressions take their places at the front of the structure. The formal semantics of ferns are presented in two ways which are proved to be weakly equivalent in that any interpretation under one may occur under the other. Examples presented include an _or_ operation which converges when any of its disjuncts

---

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS75-06678 A01, MCS75-08145, and MCS77-22325.

are true, a *merge* of arbitrarily many streams, and an airline reservation system based on a rudimentary interrupt handler monitoring an arbitrary number of input devices.

Keyword and Phrases: asynchronous recursion, fern, *amb*, multiset, bag, *frons*, *cons*, list, sequence, prefix*, stream, powerdomain, nondeterminism, parallel processing, suspension, lazy evaluation, call-by-need, call-by-delayed-value, unpredictable ordering, interrupt, semaphore, monitor, serializer, fair merge, arbiter, breadth-first search, generator.

CR Categories: 4.20, 4.32, 4.34, 4.13, 4.35.

Introduction

An example long standard in computer architecture is the "interrupt" mechanism by which a peripheral device may communicate with the central processor. The concept has been more recently formalized as semaphores [10], monitors [3,29], or serializers [27] in order to allow closely coupled processors to share storage resources without interference. These constructs are only a tool for grappling with the timing problems. Systems programmers are still responsible for correctly programming very low-level code which uses these tools. Theirs has become a notoriously occult skill [11,23] which is unavailable to user programmers who would now like to take advantage of asynchronous and unpredictable program behavior by using many (parallel) processors to solve a single problem. One should not be surprised that the users do not have programming talents or the languages at their disposal which would allow them to express parallelism for any but the most obvious cases.

We have been working with applicative programming, a style in which the only form of expression is the application of a function to an argument (perhaps structured as an argument list), as an approach to these problems. At the elementary level, our style of expression is quite familiar:

```
or:<FALSE TRUE TRUE> = TRUE .
```

In this case the disjunct function is applied, by the colon, to the argument which is a sequence of three truth values. (Angle brackets indicate the sequence of what occurs between them; parentheses indicate their actual appearance following evaluation.) Less familiar might be the definition of functions

using application recursively as the only "control structure"

```
factorial:n ≡
  if iszero:n then 1
  else product:<n factorial:pred:n>
```

is probably a familiar definition, but one can express

```
or:disjuncts ≡
  if null:disjuncts then FALSE
  elseif first:disjuncts then TRUE
  else or:rest:disjuncts
```

just as precisely and more easily than with assignment statements and loops. We believe that either of these definitions is more expressive and easier to compose than similar programs written with the more standard tools of iteration and assignment statements [5]. The problem of efficiency, often raised against such examples, is not at issue here.

Others have approached practical problems with similar abstract tools. Some [2, 8] consider applicative styles from the perspective of hardware and machine architecture. Backus [1] and Hewitt and Smith [28] have considered such expression from the perspective of software, and Burge [4] and Kahn and MacQueen [31] have particularly considered applicative programming in dealing with unbounded structures. The plethora of results on program transformations which assume applicative programs as the source [5, 52, 53, 57, 58] suggests that although this is not only metal for casting our programs, at least it is ductile and malleable. Programs so expressed may be reshaped to suit local hardware and demands for efficiency.

This style of expression is familiar in the form of lambda calculus [6] and combinators [7] and programming languages like ISWIM [38,4] and pure LISP [39]. We had already derived results uncovering unforeseen parallelism [15] in fifteen-year old LISP code written with standard sequence constructor functions [14] or a simple extension of functional application to sequence of functions [18]. These appeared to subsume expression of the "most obvious" cases of parallelism quite easily; and our attention turned to expressing the less obvious parallelisms, which seemed to require mystical programming techniques, in an applicative fashion.

In McCarthy's formal axiomatic sense [39] the sequence S = <a b c>, constructed with the sequence constructor <u>cons</u>, should allow each element of the sequence to be accessed independently of the convergence of the other two elements. We use the integer i as a function to access the ith item of a sequence.

1:S = a, 2:S = b, and 3:S = c,

each regardless of whether the other two elements diverge. The axiomatization can be implemented by arranging that no element of a sequence is ever evaluated until it is accessed, e.g. via the lazy evaluation scheme [25], call-by-need [55], or call-by-delayed-value [54] protocol for function invocation. It may be effected by replacing a value by a <u>suspension</u> of that value until it is probed by an access function, which is only indirectly invoked by the need of some sequential output device to display the contents of a suspended result [16].

*****

Could there be a way to construct an unordered structure U = {a b c} suspended in a manner similar to the mechanism used by cons? The idea is that 1:U would return the first convergent value of the multiset (we call U a multiset since two elements may converge to the same value yet its size is unaffected [34]). The 2:U returns the second convergent value if there is one; and 3:U returns the third and is, therefore, defined only when all three elements converge. The answer is "yes"; and the remainder of this paper develops a constructor, frons, which is used in a manner most similar to the sequence constructor defined by McCarthy and used fluently by all LISP programmers.

*****

This new constructor allows the building of a structure whose order is not preordained by the programmer, but as accessed at runtime, that structure takes on a total ordering as its prefix is accessed. The order will be determined then by the necessary convergence of those items which are successfully accessed. If an item in the specified multiset cannot converge because an asynchronous event has not occurred-- because a real-time signal has not been received-- then that item is simply excluded from candidacy for "firstness". When all real-time events on which its evaluation depends have occurred, that item at last becomes a candidate to be first; and eventually under a fair implementation it will be found to be the first item of the multiset of some suffix. As the order of a multiset is uncovered that multiset becomes indistinguishable from a list!

With a single new constructor function, the naturally extended accessing functions, _first_ and _rest_, and applicative programming as _the_ control we have developed an expression for several programming techniques. For instance, breadth-first evaluation may be specified by choosing the first of a (perhaps infinite) multiset of the possible answers:

or:{and:{a b c} and:{d e f} and:{g h}} .

It may not matter in this example if _a_, _c_, and _g_ diverge. If it turns out that _b_, _e_, and _h_ are FALSE, then the entire expression evaluates to FALSE. If _d_, _e_, and _f_ are TRUE, then so is the expression. If convergence of even these values must await some unknown event, then the evaluator proceeds until the requisite event(s) occurs, since there can be no other value until then. Semaphores or interrupts are drawn implicitly into the computation by the evaluator, since the programmer need not specify their direct control over this expression. When the necessary event occurs, an indirect effect may be to cause convergence of the logical expression above; but the creator of this expression need not be aware of how "control" of the machine has returned to yield his value-- of why his computation suddenly awakens and proceeds. As we shall see, distributed evaluation is necessary to the proper implementation of the constructor _frons_; and it appears that _frons_ allows us the power to express less obvious parallel processing now familiar to systems programmers.

The major dilemma facing the real-time programmer is handling asynchronous and indeterminately occuring external events using a programming style built upon deterministic and synchronous interprocess communication. (At the microcode level each register might be viewed as a process.) By embedding the asynchronous events within a multiset and specifying the behavior of access to that structure, we have cut the synchronous/deterministic stone in a new and different fashion. We emphasize a structure, whose access pattern is undefined *a priori*, but whose *a posteriori* behavior is exactly that of a sequence; it becomes a sequence in an unpredictable way. The programmer need only ascertain that his program gives valid results no matter which of the possible sequences appear. It will become evident in the examples that this kind of correctness is easier to assure than the other [11,23,36,44]. By removing any concern for order from the programmer, we free him to specify what he wishes to do with the results without worry for how the results were derived. The intellectual effort occurs when he specifies the content of his multiset (which is a statement of what can happen) and when he specifies the use of each element as it is uncovered (which is a statement assuming something has happened). He need not worry about *how* it happened as does the "unpredictable" programmer who must make sure that all computation paths are valid [23].

In a sense the style is asynchronous programming as opposed to "perceived nondeterministic" programming. The programmer need not worry about flow of control on the arrival of external

signals since there is no explicit flow of control. Therefore, the programmer who uses multisets to express what others recognize as indeterminacy is working with a concept which is more natural since it is freed of the discrete time assumption of actual computers. To him the evaluation of all the unevaluated elements of a multiset proceeds asynchronously, and the evaluator is, therefore, free to treat such evaluations as independent parallel processes.

The remainder of this paper is in five parts. The next section presents a semantics for indeterminately ordered structures as ferns, a restriction of forests. This development allows a graph-like visualization of the unordered structures intended, and justifies the practical terminology we use. It is followed by a brief section presenting additional tools needed in the third section. In order to present the user's vier of <u>cons</u> and the new constructor, <u>frons</u>, the third section is the program of an airline reservation system which handles many active or inactive agents' terminals. The fourth section offers a concise structure semantics for expressions involving <u>cons</u>, <u>frons</u>, <u>first</u>, <u>rest</u>, and <u>null</u> which will be more useful than the earlier fern semantics. We prove that the two semantics are similar (weakly semantically equivalent). The conclusion argues that the user's experience with data structures will strengthen that similarity in practice because the programmer tends not to create duplicate structures; rather he shares extant ones. Ongoing work is also described there.

Fern semantics

A relation E on a set N is a subset of $N \times N$; the elements of N are called nodes and those in E are called edges. A partial order is a relation, E, such that

(i) If $(x,y) \in E$ then $(y,x) \notin E$ (asymmetry).

(ii) If $(x,y) \in E$ and $(y,z) \in E$ then $(x,z) \in E$ (transitivity).

A partial order is irreflexive because if E is asymmetric then (x,x) may not be in E. A relation is ancestral if $(x,z) \in E$ and $(y,z) \in E$ together imply that either $(x,y) \in E$, or $(y,x) \in E$, or $x = y$. Pairing a set with a relation on it (N,E) yields a digraph; when the relation is a partial order it yields a partially ordered set. A total order or a chain is a partially ordered set (N,E) in which for all $x,y \in N$ either $(x,y) \in E$ or $(y,x) \in E$ or $x = y$. A node $x \in N$ is minimal in the digraph (N,E) if $(y,x) \notin E$ for all $y \in N$. A chain is rooted if it includes a minimal element. A forest is an ancestral partially ordered set in which every embedded chain is rooted. When we draw the directed graph of a forest, below, we shall omit edges implied by transitivity in the associated partial orders; the resulting graph (Hasse diagram) appears like a bunch of trees. Elsewhere the term "forest" applies to such a graph, rather than to one which includes the edges implied by transitivity and included in our definition; the difference is slight.

There is no explicit need in such figures to label the nodes, because the circles representing the nodes themselves are sufficient to represent the set N; we may, however, elect to label each node according to some labelling scheme which augments the forest to a labelled forest. A labelling of N is a function from N to some domain D. A labelled forest is a triple (N,E,V) where (N,E) is a forest and V is a labelling of N.

Definition: A fern is a digraph (N,E) where

(i) There is a partition of N into three sets: B, L, and S called the bud set, the leaves[1], and the sprout, respectively.

(ii) B is a singleton whose element, called the bud, is the root of the sprout: $B \times S \subset E$; $N \times B \cap E = \emptyset$.

(iii) L is isolated: $(L \times N \cup N \times L) \cap E = \emptyset$.

(iv) The restriction of E to S, $R = E \cap S \times S$, is such that the sprout forms a fern: (S,R) is a fern.

Several observations follow from the definition of a fern. Most are apparent from studying Figure 1. A fern must have at least one node (the bud) but the partitioning into bud set, leaves, and sprout need not be unique when there are no edges.

---

[1]We use the term leaf to mean something different than a terminal node.

The observation that every fern is a forest explains the choice of the term "fern". A fern is like a forest, but the branching is very spindly. One is reminded of the derivation trees of linear grammars [24], or the branching pattern of the plant <u>equisetum</u> (Figure 2) or a Roman fountain (Figure 3). Following upon this observation we define a <u>labelled fern</u> to be a labelled forest whose underlying forest is a fern. Figure 1 illustrates a labelled fern. The symbol "$\perp$", read "bottom" is used there to indicate that the labelling function does not converge (in the recursive-function-theoretical sense) or does not evaluate to a value in the label domain D (as opposed to the extended domain D^+ [41, p. 358]). The labelling function in Figure 1 may be perceived, therefore, either as a partial function from N to D or as a total function from N to D^+. The only labelled ferns which we shall consider here either are infinite or have at least one node whose label is $\perp$.

<u>Definition</u>: A <u>sequence</u> is a fern whose partition according to the fern definition is such that $L = \emptyset$ and such that either $S = \emptyset$ or (S,R) is a sequence.

A sequence is a fern with no leaves whose non-trivial sprout forms a sequence. A sequence defines a total order on its set. We may extend this concept to define <u>labelled sequences</u> as a special case of labelled ferns. Labelled ferns may be finite or infinite. In order to develop the specific concept of a finite fern we introduce the concept of a <u>null</u> fern.

Definition: NIL is the labelled fern $(\{q\},\emptyset,\{(q,\perp)\})$.

NIL is the most trivial fern; an alternative definition (from the perspective of the labelling function being partial recursive) is that NIL = $(\{q\},\emptyset,\emptyset)$.

Our intent in defining a fern is to characterize all data structures as labelled ferns. We, therefore, shall use the terms "fern" and "sequence" for labelled fern and labelled sequence, respectively. Since we shall embed these data structures in a computational environment, we confront the need for a semantics for ferns under some interpretation function, I, which we presume to be already defined on non-ferns. (Non-ferns include elementary items, elementary expressions which are built using primitives not explicitly defined here, and function invocation which may create environments that are also embedded in I.) As we specify the interpretation of ferns we shall arrange that the labelling function carry the interpretations of its elements. The absence of a meaningful labelling in NIL is therefore of no consequence, since we shall perceive NIL as a structure to be devoid of content.

We now can begin to define our fern constructor, fons, and present its semantics under an interpretation function I, which determines the labelling functions. We define it in two steps fons_T and fons_F with T and F indicating Boolean values.

Definition: If $I[\![f]\!] = F = (N,E,V)$ is a labelled fern, we define the extension of I to $\underline{fons}_T$ to be

$$I[\![fons_T(x,f)]\!] = (N \cup \{q\},\ E \cup \{q\} \times N,\ V \cup \{(q, I[\![x]\!])\})$$

where q is some new node not in N or elsewhere in I. The set E is time-dependent in the sense that it contains all edges defined *a priori* or *a posteriori* (as a result of side-effects from EUREKA as set forth below).

The minimal elements of a fern are the bud and the leaves. the effect of $\underline{fons}_T$ is to extend the labelled fern F by including a new element whose value is that of $I[\![x]\!]$. Furthermore, that element will be the unique minimal element in the resulting fern *a priori*.

If a fern is built from NIL using only $\underline{fons}_T$, then the interpretation of the result under I must be a sequence. Thus the effect of $\underline{fons}_T$ is that of the well-known constructor cons which builds lists like in LISP [39]. From this observation [14] and from results on parameter passing mechanisms [54, 55] we make the important observation that the interpretation $I[\![fons_T(x,f)]\!]$ defined above converges whether or not $I[\![x]\!]$ (or $I[\![f]\!]$) does[2]. This will be particularly important in defining the interpretation of the rest function below which, as we shall see, is defined independently of x.

[2] The same remark applies to $\underline{fons}_F$. In a lattice theoretic approach [48] establishing that a datum is constructed using $\underline{fons}_\beta$ is therefore sufficient to prove that it is not $\perp$.

Because finite sequences are so useful in list processing, we shall specify a special notation for them.

Notation: $I[\![<>]\!] = NIL$;

$$I[\![<x_1\ x_2\ \ldots\ x_k>]\!] = I[\![fons_{\mathcal{T}}(x_1, \iota<x_2\ \ldots\ x_k>)]\!] .$$

So far the ferns which result from these interpretations are not useful in and of themselves. We specify three functions, null, first, and rest, which the programmer may use to probe these ferns, and then specify how the fern is presented as a list if it is to be viewed in its physical manifestation.

First we shall define a non-deterministic transformation, EUREKA, on ferns. The effect of EUREKA is to locate some element which is minimal in the fern. If the fern has leaves then EUREKA refines the fern by adding edges so that the chosen node becomes the unique minimal element. This is called promotion. EUREKA chooses the unique minimal element by finding a node q such that $q \in L \cup B$ and $V(q) \neq \bot$. Thus, EUREKA has a side-effect of augmenting the definition of the fern in question a posteriori, and all these edges--although they may be added later-- appear to have been in that fern all the time. If the fern has no leaves, then EUREKA leaves it unchanged.

Definition: If $F = (N,E,V)$ is a labelled fern then

$$EUREKA(F) = \begin{cases} F \text{ if } L = \emptyset; \\ (N,\ E \cup \{q\}\times(N-\{q\}),\ V) \text{ if } L \neq \emptyset \\ \text{where } q \in N \text{ is minimal in } E, \text{ and } V(q) \neq \bot \\ \text{and all interpretations over F are thereby} \\ \text{augmented}^{3} \text{ to equal EUREKA(F).} \end{cases}$$

---

[3]Let $F = (N,E,V)$ and $F' = (N',E',V')$ and if $q,r \in N \cap N'$ then $(q,r) \in E$ implies $(q,r) \in E'$. These edges are added a posteriori as suggested earlier.

After EUREKA has been called once, its invocation on the <u>same</u> fern thereafter must behave as the identity function. The selection of the minimal element q, which will be promoted to be the unique minimal element in F, is deterministic when the fern has no leaves, and once EUREKA has been successfully invoked no leaves will remain. In the case, however, that there is at least one leaf EUREKA is a mapping strict [54] in the label of some leaf or of the bud[4]; that is, EUREKA cannot converge until some such label converges.

<u>Definition</u>: If I⟦f⟧ = F = (N,E,V) is a labelled fern then

$$I[\![null(f)]\!] = \begin{cases} TRUE, \text{ if } |N| = 1; \\ FALSE, \text{ otherwise.} \end{cases}$$

and if q is the unique minimal node in (N,E) after EUREKA(F) then

$$I[\![first(f)]\!] = V(q).$$

$$I[\![rest(f)]\!] = (M,\ E \cap M{\times}M,\ V \cap M{\times}D^{+}) \text{ where } M = N - \{q\};$$

(i.e. F with q removed).

These are the <u>only</u> primitive operations defined on ferns! Even when a person asks to see a fern, he may at best ask to see a copy of the linear traversal of that fern using <u>null</u>, <u>first</u>, and <u>rest</u>. Such a linear traversal will be a topological sort [33] of the partial order originally embedded in the fern and would appear as a list.

[4] A function is strict in its $i^{\underline{th}}$ parameter if divergence of the $i^{\underline{th}}$ argument implies divergence of the function. We are extending this concept to promotion of ferns with labels acting as arguments.

Definition: The list of a fern is the list of characters defined by the PRESENT operation below ($\circ$ denotes string concatenation):

$$\text{PRESENT}[\![x]\!] = \begin{cases} \text{print image of } I[\![x]\!], \text{ if } I[\![x]\!] \text{ is an elementary item;} \\ \text{"("} \circ \text{TRAVERSE}[\![x]\!], \text{ if } I[\![x]\!] \text{ is a labelled fern.} \end{cases}$$

TRAVERSE is the traversal of I⟦x⟧ up through the right parenthesis:

$$\text{TRAVERSE}[\![f]\!] = \begin{cases} \text{")"}, \text{ if } I[\![\text{null}(f)]\!]; \\ \text{PRESENT}[\![\text{first}(f)]\!] \circ \text{TRAVERSE}[\![\text{rest}(f)]\!], \text{ otherwise.} \end{cases}$$

Thus far we have defined labelled ferns in general, but we have given fern semantics only for labelled sequences. Moreover, we have two ways to specify sequences; using the constructor $fons_T$ (i.e. cons) and using the angle-bracket convention for finite ferns. In both cases the PRESENT operation produces a list which is little more than a homomorphism of the interpretation down the sequence; EUREKA is deterministic on such structures.

The definition of first and rest, however, already handles a fern which is not a sequence.

Definition: A multiset is a labelled fern whose partition according to the fern definition is such that $S = \emptyset$[5].

The only multiset which is a sequence is NIL. Thus NIL suffices to represent the empty multiset as well. We note that the partition of a non-empty multiset (which determines the bud set) is not unique. If every element in a labelled forest is minimal then it is necessarily a multiset.

---

[5]This definition is consistent with Knuth's usage [34]; note that the labelling of a fern allows duplicate values. Waldinger [56] (see also [47]) uses the term "bag" for a structure which appears similar to our multiset. There are significant differences however. A bag may only have a finite number of elements. Every element of a bag must be computationally convergent (in D) whereas we allow labels on elements of a multiset to diverge (through a divergence of the labelling function).

We complete the definition of the general fern constructor in order to allow the construction of multisets.

Definition: If $I[\![f]\!] = F = (N,E,V)$ is a labelled fern then

$$I[\![\underline{\text{fons}}_F(x,f)]\!] = (N \cup \{q\},\ E,\ V \cup \{(q, I[\![x]\!])\}\ )$$

where q is some new item not in N.

The set E is time-dependent as it was in the definition of $\underline{\text{fons}}_T$.

Thus, we have interpretations on $\underline{\text{fons}}_\beta$ for either of the two Boolean values for β. We thereby have defined our two fern constructor functions as just one if we perceive β as a third argument which specifies whether or not edges are to be added into the fern a priori as it is constructed. The interpretation is that when $\beta = T$ then edges are specified as an element is added to the fern; when $\beta = F$ no new edges are added, leaving open the possibility that they (or their inverses--not both) may be added at some later time. Just as $\underline{\text{fons}}_T$ is called cons by programmers, in the example section we shall rename $\underline{\text{fons}}_F$ to frons since it adds leaves to the fountain-like ferns.

A notation for multisets, analogous to the notation for sequences (using angle brackets), is quite useful.

Notation: $I[\![\{\}]\!] = \text{NIL}$;

$$I[\![\{x_1\ x_2\ \ldots\ x_k\}]\!] = I[\![\text{fons}_F(x_1,\ \{x_2\ \ldots\ x_k\}\ )]\!]\ .$$

Because we allow labelled ferns which are neither multisets nor labelled sequences it should be obvious that there is no restriction on the way that the $\underline{\text{fons}}_\beta$ constructors may be

interweaved. At this time we do not know any obvious uses which would require the programmer to specify such hybrid ferns. Nevertheless, we extend the bracketing notation to include such hybrids by using vertical bars.

<u>Notation:</u> $I[\![\{x_1 | \; x_2 \; x_3 \; \ldots \; x_k\}]\!] = I[\![fons_T(x_1, \; \{x_2 \; x_3 \; \ldots \; x_k\} \;)]\!]$

Thus the presence of a following bar indicates that $\underline{fons}_T$ is the necessary constructor; its absence indicates that $\underline{fons}_F$ is the constructor. Hence <a b c d> = {a| b| c| d|} under I. The effect is that the vertical bar follows a necessary choice for a bud.

Although the programmer appears to have little need to use the bar notation, ferns which are neither sequences nor multisets may arise as internal structures as EUREKA works its intent on multisets. For instance, Figure 1 might arise as the result of interpretations on the multiset {jj mm} to which gg, hh, ii, kk, and ℓℓ were subsequently added, and later to which aa, bb, cc, dd, ee, and ff were added. Here we assume that I⟦aa⟧ = AA, I⟦bb⟧ = BB, I⟦cc⟧ = CC, I⟦dd⟧ = DD, I⟦ee⟧ = EE, I⟦ff⟧ = FF, I⟦gg⟧ = GG, I⟦hh⟧ = HH, I⟦ii⟧ = ⊥, I⟦jj⟧ = JJ, I⟦kk⟧ = KK, I⟦ℓℓ⟧ = LL, and I⟦mm⟧ = MM. Using the brace-bar notation Figure 1 might also have arisen from

{cc ee bb dd aa ff gg| kk ii ℓℓ hh jj| mm}

or {ee cc dd bb aa ff gg| ii kk ℓℓ hh jj| mm}.

For comparison, Figure 4 arises from the sequence <aa bb cc dd> and Figure 5 illustrates the multiset {ee ff gg hh} under these assumptions.

Let us consider the _a posteriori_ effects of EUREKA on the graph representation of the fern F of Figure 1. Suppose that we find that I⟦first(f)⟧ = CC (where I⟦f⟧ = F). The implication is that EUREKA selected the node associated with CC (i.e. cc) as its "chosen one" from among those associated with aa, bb, cc, dd, ee, ff, and gg; it was chosen because its label converged to CC. EUREKA adds edges to F to yield the labelled fern (which is the transitive completion of the one) in Figure 6. Henceforth and previously F equals this fern!!

It follows from Figure 6 that we could describe I⟦rest(f)⟧ as {aa bb dd ee ff gg| hh ii kk ℓℓ jj| mm} represented there by the fern excluding the node associated with CC and I⟦f⟧ will now be {cc| aa bb dd ee ff gg| hh ii kk ℓℓ jj| mm}. Extending the example further we might consider the effect of EUREKA on F after discovering that I⟦first(rest(f))⟧ = GG. Figure 7 illustrates the effect of adding more edges to F. In this case a new level is not introduced into the picture (viewed as a tree); instead the new edges have the effect of pushing leaves from one level up to the next because it is the bud which EUREKA has promoted. Figure 7 also indicates how the accessing--the use-- of F slowly coerces it into a sequence. At this point only the first two elements of F have been probed, but the structure has been augmented so that those two probes have reproducible results. Another interpretation of Figure 7 is the situation after the PRESENT operation has traversed the first two elements of F, but not yet the remainder; the list which it presents appears as "(CC GG ... ". Any picture of a fern other than (a prefix to) the list it represents would anticipate eventual effects of EUREKA.

Implementation of the first invocation of the function _first_ or _rest_ (i.e. EUREKA when there is more than one element) requires some sort of parallel evaluation strategy over all label computations on elements which are minimal in the fern. The implementor is free to choose any strategy he likes, so long as no such computations are neglected. (We have not excluded the possibility of an infinite number of minimal elements.) A correct implementation converges to the first convergent minimal label of a fern (whenever there is but one) and a good implementation will converge fairly rapidly. For instance, in the case where the fern, F, is a multiset of external events (like console keystrokes), only one of which actually occurs, we would expect first:f to converge very quickly after a key was actually pushed.

As an example of such a situation, consider the disjunction of three such values, any one of which may be undefined, dependent on external "force fields", or sufficiently defined but extremely difficult to compute. In spite of these adversely defined values, some one of the three is very simple to compute and is, in fact, TRUE. The problem is that we don't know which one it is! Consider the interpretation of the expression "or:<x y z> in which the three arguments are structured as a sequence. The standard code for _or_ in the introduction requires that the first argument, _x_, converges before the rest are accessed. If

the arguments are structured as a multiset, however, there is no necessity of evaluation in any particular order. The interpretation of "or:{x y z}" provides that evaluation of all three disjuncts proceeds simultaneously--that is, evaluation of any one does not preclude progress on evaluation of any other. The first value to converge becomes the _first_ element in the multiset (actually, now a hybrid), and so the code for _or_ above now imposes a "call-by-convergence" protocol. The first disjunct which produces a TRUE label will terminate computation of the disjunction. Until then the evaluation of all disjuncts proceeds; so it doesn't matter which of the three eventually becomes TRUE. If one of them does then the disjunction itself is well-defined.

We close this section with the remark that the domain of the labels has thus far remained unspecified. We could specify that domain to include ferns and elementary items, so that the structures defined here would have depth as well as width when traversed, say by the PRESENT operation. The possibility of a label being a subfern thus generalizes ferns to include the classic data structures like Lists [33] and trees. One should not confuse the sprout infrastructure, which has implications for the eventual order of the list which a fern represents, with the infrastructure of a subfern which is but one label in that underspecified ordering.

Additional tools

Before we turn to practical applications of the multiset, we return to some elementary implementation considerations. The first problem is to reconcile our use of brackets in the introduction with our definition of them as a shorthand for constructing ferns above. Second is the introduction of some elementary primitive functions. Third is the definition of a strictify primitive as a two argument identity function which is strict in both arguments.

The interpretation of an angle-bracketed sequence in the previous section is a fern, specifically a labelled sequence. The interpretations of first and rest on such a sequence yield the interpretations of the respective elements in the same way that we would expect such probings to behave on the angle-bracketed sequences of the Introduction. This is true because the underlying fern is a total order; and there can be no change in the order of the values seen by first and rest (i.e. EUREKA would always behave as an identity function). We depend on the interpretation, I, to assure that the values are unchanged as well. The definition of braced multisets in the previous section similarly meets our design for such structures anticipated in the introduction.

We have already seen the use of angle-bracketed and braced argument lists in the example of disjunction at the end of the previous section. This notation will be used extensively in the code of the following section. In all cases it is consistent with the notions of structure outlined in the Introduction and reconciled here so that the interpretations of the user's code first:x, rest:x, null:x, cons:<x y>, and frons:<x y> are respectively $I[\![first(x)]\!]$, $I[\![rest(x)]\!]$, $I[\![null(x)]\!]$, $I[\![fons_T(x,y)]\!]$, and $I[\![fons_F(x,y)]\!]$.

Definition: The predicate same is defined by the interpretation

$I[\![same(x,y)]\!] = (I[\![x]\!] = I[\![y]\!])$ where $I[\![y]\!]$ is always an elementary value.

As mentioned above we shall use the syntax, same:<x y>, rather than the parenthesis notation here. The other primitives are arithmetic, and we define them less formally. We assume the functions, succ and pred, and the predicate iszero which respectively return the successor, the predecessor, and the "zeroness" of their argument. As an example of their use, we define a function which yields an infinite sequence of the integers greater than or equal to i:

```
integers:i ≡ cons:<i integers:succ:i> .
```

We can then define the sequence of all natural numbers by

```
naturals ≡ integers:0 .
```

Finally we define the strictify primitive. Because we shall be dealing with structures whose order depends on the convergence of (the labels of) their elements and because the values of those elements themselves may be structures, we require a convenient notation for passing the convergence or divergence up through those structures to the computations of the (labels of the) elements of a multiset.

Definition:

$$I[\![\text{strictify}(x,y)]\!] = \begin{cases} I[\![y]\!] \text{ if } I[\![x]\!] \neq \bot; \\ \bot, \text{ otherwise.} \end{cases}$$

A few examples will suffice to demonstrate the generality of strictify. Each of these examples uses strictify to recover familiar semantics of some older functions. Consider valuecons as the function strict in both arguments as normally implemented using call-by-value protocol in LISP or Landin's prefix* [37,4] which is similar but only strict in the first argument. Both may be implemented in terms of cons which suspends evaluation of both arguments until probed with first or rest [14]:

```
valuecons:<x y> ≡ strictify:<x strictify:<y cons:<x y>>>;
prefix*:<x y> ≡ strictify:<x cons:<x y>> .
```

Finally we consider restricting a parallel conditional primitive [42,20] to the more common sequential conditional [48,54]:

```
seqif:<pred then else> ≡ strictify:<pred
                                    parif:<pred then else>> .
```

The airline reservation system

In order to demonstrate the facility of nondeterministic programming with the new constructor *frons*, we present a few example programs leading to an airline reservation system which solve problems of nondeterminism in an applicative style. *In reading these examples one should notice how nondeterminism is isolated into the data structure, so that the program is rather simple!* First we consider the problem of flattening a multiset of streams [37] (i.e. sequences constructed by *prefix**) into a list. In this example the argument, a matrix with two unbounded dimensions (i.e. the number of rows and the number of elements in each row may be infinite, in which case the first two lines of *merge* are meaningless).

```
merge:M ≡
  if null:M then M
  elseif null:first:M then merge:rest:M
  else cons:<first:first:M
             merge:frons:<rest:first:M rest:M>> .
```

The use of the two constructor functions in *merge* is particularly interesting. Assuming that the value for M is defined appropriately, we can interpret the four possible substitutions of the two constructors in those two positions. If both were *cons*, then *merge* would append [40] all the rows of M in the order they are presented in M; if the first row of M is infinite then that row would be copied. With the constructors as in the definition of *merge*, the effect is to interleave the various rows of M, so the order of each one is preserved, but elements from other rows may

be interspersed in the final result. If both constructors were frons, the effect would be to allow any shuffling [32] of all the elements of the array as an ordering in the result. (In the unusual case that the first constructor were frons and the second one were cons, a similar shuffle would result; but elements in the result would be restricted to rows only up through the first infinite one in M.)

The interleaving behavior is what we desire for the next example. We would like to write a nondeterministic input driver for a time-sharing system. Specifically, we want to solve the input problem for the airline reservation system [59, 9]. In that problem we have an arbitrary number of remote agents' terminals each producing an infinite stream of characters. Each stream forms a row of an input matrix. Thus every row is infinite (as time passes); and there are an indeterminate number of rows (new terminals may be activated at any time). The problem is to write an applicative program which will accept these characters as soon as they are typed (regardless of the inactivity of an other terminal) and interleave them into a single input stream with each character identified according to its source.

We require an auxiliary function which will transform a file --a sequence of characters-- into a sequence of pairs --a character and the signature of the file. Furthermore, that sequence of pairs, and all its suffixes, should be strict in the convergence of their first characters. This strictness precludes convergence of such sequences until their first character has been typed at the corresponding remote terminal.

```
identify:<file id> ≡
  if null:file then file
  else strictify:<first:file
                cons:<<first:file id> identify:<rest:file id>>> .
```

(Even though identify specifies a full computation over file, the reader should satisfy himself that each step is suspended until it is needed [14].)

It is the multiset of identified files which must be merged. Let us assume that files is a sequence of the sequential files to be interleaved. Then we may invoke the function fanin upon files and naturals in order to generate the desired stream of agents' communication:

```
fanin:<files signatures> ≡ merge:identifyall:<files signatures> ;

identifyall:<files signatures> ≡
  frons:<identify:<first:files first:signatures>
         identifyall:<rest:files rest:signatures>> .
```

The identifyall function is used to convert the sequence of files and signatures into a multiset of "strictified-identified" files. Thus, the application of first in merge can only yield a result from an active terminal.

For example, suppose we have three files being generated in real time by three different terminals. A file being so typed is viewed as an infinite list of characters which becomes finite only when the wire over which it communicates is cut. Of course, that may happen only *after* the computer running *fanin* has been decommissioned; so *fanin* does not allow for finite files. Suppose that Jefferson types file-0 as (W H E N _ I N _ T H E _ C O U R S E _ O F _ H U M A N ... and Lincoln types file-1 as (F O U R S C O R E _ A N D _ S E V E N _ Y E A R S _ A G O ... and Rip Van-Winkle is at terminal 2 typing nothing at all, then fanin:<<Jefferson Lincoln Van-Winkle> naturals> might return

((W 0)(H 0)(E 0)(F 1)(O 1)(N 0)(U 1)(_ 0)(R 1)(I 0)(S 1)...

or any of several other shuffles of the two active files. Given a finite number of files, the function *fanin* accomplishes the same work as Kosinski's *arbiter* [35].

Just as *fanin* merges a sequence of files into a single list, there is an inverse function which takes the result of *fanin* (i.e. a list of character-index pairs) and produces the original list of files. We call this function *fanout*. It has the property that "fanout:fanin:<files naturals> = files. The intended use of *fanout* is to distribute results of a multiple valued function to several devices [17]. In many cases that function, in turn, calls *fanin* to assemble its input into an ordered stream. The function *buildup* inserts a new file in the indexth position of a list of files. The new file is the old file with a character *consed* onto the front of the file.

```
fanout:pairs ≡ buildup:<first:pairs fanout:rest:pairs> ;

buildup:<<char index> files> ≡
  if iszero:index then cons:<cons:<char first:files>
                             rest:files>
  else cons:<first:files
             buildup:<<char pred:index> rest:files>> .
```

(N. B. that buildup uses a structured formal parameter [28].) The result of fanout may be passed to I/O media, but if each file of the result is tied to a sequential device, the results appear overlapped with the input [16]. Because all files are here presumed to be unbounded, we have no null tests.

We are at last prepared to write an airline reservation system [12] as a classic problem representative of real-time programming systems. The essence of this problem lies in its need for a shared data base, the planes' bookings, and for the organization of simultaneous requests from booking agents. For each of these requirements the use of a protected critical section (using a semaphore [10], a monitor [3, 29], or a serializer [27]) is generally required. We dispense with the issue of a shared data base by returning a "changed" data base as one of the results of the function which changes the data base; we have discussed elsewhere [17] how these "suspended copies" of a file guarantee its integrity. The issue of simultaneous requests has been solved by fanin, although we have yet to guarantee that all booking agents will receive attention while the system remains completely busy. This issue of fairness is addressed elsewhere [21].

Our solution works with just one plane and an arbitrary number of agents. (Our airline is adding agent terminals all the time.) Generalization to a full system is not a formidable task. The requests from each agent (individual at a terminal) are structured as a sequential file; and the responses to each agent are also structured as a sequential file. The parameters for airlinereservationsystem are the request files and capacity of the plane. This is the essential structure of the system:

```
airlinereservationsystem:<files capacity> ≡
  fanout:genresponses:<fanin:<files naturals> capacity> .
```

The function genresponses takes as parameters a sequence of character-index pairs and the seating capacity of the plane. The character of each pair is the request. If the request is R (C) then it is a reservation (cancellation). If the request is a different character, then it is an unknown command. If the request is a reservation and there are no vacancies, then the response is F (full); otherwise the request is echoed implying success. If the request is a cancellation and there have been no reservations, then the response is E (empty); otherwise the request is echoed implying success. If the request is not C or R, then the current number of vacancies, perhaps stale information, is returned indicating no action. This description of the airline reservation system is also the outline of the auxiliary function gen. The case where a successful reservation (cancellation) is made is handled by declaring one

of the seats occupied (unoccupied) with "pred:vacancies" (succ:vacancies). The function genresponses (i.e. gen) only builds sequences. More nondeterminism can be realized by building the structure of responses with frons or by using parallel evaluation of conditional expressions [20].

```
genresponses:<pairs capacity> ≡
  gen:<first:pairs rest:pairs capacity capacity>;

gen:<<request index> pairs vacancies capacity> ≡
  if same:<request R> then
     if iszero:vacancies then cons:<<F index> gen:<first:pairs
                                                   rest:pairs
                                                   vacancies
                                                   capacity>>
     else cons:<<R index> gen:<first:pairs
                               rest:pairs
                               pred:vacancies
                               capacity>>
  elseif same:<request C> then
     if equal:<vacancies capacity> then cons:<<E index> gen:<first:pairs
                                                             rest:pairs
                                                             vacancies
                                                             capacity>>
     else cons:<<C index> gen:<first:pairs
                               rest:pairs
                               succ:vacancies
                               capacity>>
  else cons:<<vacancies index> gen:<first:pairs
                                    rest:pairs
                                    vacancies
                                    capacity>> .
```

Structure semantics

In this section we present an alternative and more formal semantics for the constructors cons and frons (here again called $fons_T$ and $fons_F$). This semantics will appear to be a purely functional presentation, except for the use of the nondeterministic operator amb. McCarthy [39] proposed amb an an "ambiguity" operator of two parameter which is strict in neither:

Definition:

$$AMB(x,y) = \begin{cases} x \text{ if } x \neq \bot; \\ y \text{ if } y \neq \bot; \\ \bot, \text{ otherwise.} \end{cases}$$

AMB in and of itself is not a continuous function [48] because it may not return the same result when applied repeatedly to the same arguments (but a continuity argument is available if we consider powerdomain constructions [45,51].) This would be a serious difficulty for functional semantics if we let it happen, but our use of AMB is restricted to those situations wherein AMB may only be applied to the same arguments but once. The call-by-need or call-by-delayed value protocol is assumed here [54,55,25], and AMB will only occur as an argument within a structure-building operation; since we view each structure as a different value (cf. the conclusion following) none of these invocations are repeated.

There is a strong similarity between our use of AMB and call-time choice [26]. Call-time choice may be perceived as an AMB expression passed as an argument in a call-by-need protocol. (Run-time choice may be similarly perceived as an AMB expression passed as an argument under a call-by-name protocol.) Functional

semantics are available for call-time choice under the non-repetitive convention which we assume here, so the reader is free to perceive our use of AMB in that way if he chooses.

For the purpose of this section we shall restrict the definition of <u>strictify</u> from above to a meta-language form which only returns Boolean values.

<u>Definition</u>: STRICTIFY(x,b) = b if $x \neq \bot$ and b is a Boolean value.

We now proceed to extend an interpretation function according to structure semantics. As we did in the development of fern semantics, let us assume that we have an interpretation, I', which is already defined for elementary items, elementary expressions, and function invocation (which might create new environments and thus new incarnations of I'). We shall define the extension of I' to structures in the same way that we extended I to ferns. Then, under the assumption that I on non-ferns is the same as I' on non-structures, we shall demonstrate that I = I'.

We define the set of all computational structures to include the set of atomic items and the set of structures S which are quadruples (or trivially NIL).

$D = A \cup S;$

$S = \{NIL\} \cup (D^+ \times S^+ \times \{TRUE, FALSE\}^+ \times \omega);$

where ω is the set of natural numbers; the fourth element is only used as an index on the quadruple whose effect on semantics is to guarantee uniqueness of structure.

In the following definition the occurrence of "$i \in \omega$" denotes a new integer which has not occurred in any other quadruple. As

mentioned above, for any structures indexed by such an i as a fourth element, an AMB expression as the third element is evaluated <u>at most once</u> and its evaluation is delayed until accessed. That is, as any structure is created, its fourth element is assigned a new (and different) number--say the time since Creation-- so that no other structure <u>just</u> like it is ever built again. This identification plays the same role as the uniquely created nodes (to be distinguished from the labels) in the fern semantics. The purpose of it is to allow the "same" structure to be constructed twice and yield <u>different</u> structures.

<u>Definition</u>:

$$I'[\![null(f)]\!] = \begin{cases} \text{TRUE if } I'[\![f]\!] = \text{NIL, the trivial element of S;} \\ \text{FALSE if } I'[\![f]\!] \text{ is a quadruple in S.} \end{cases}$$

```
I'⟦fons_T(x,f)⟧ = (I'⟦x⟧,
                   I'⟦f⟧,
                   TRUE,
                   i) where i ∈ ω.

I'⟦fons_F(x,f)⟧ = (I'⟦x⟧,
                   I'⟦f⟧,
                   AMB(STRICTIFY(I'⟦x⟧, TRUE),
                       STRICTIFY(I'⟦first(f)⟧, FALSE)),
                   i) where i ∈ ω.

I'⟦first(f)⟧ = FIRST(I'⟦f⟧) where
                 FIRST(NIL) = ⊥;
                 FIRST((u,v,TRUE,j)) = u; and
                 FIRST((u,v,FALSE,j)) = FIRST(v).

I'⟦rest(f)⟧ = REST(I'⟦f⟧) where
                REST(NIL) = ⊥;
                REST((u,v,TRUE,j)) = v; and
                REST((u,v,FALSE,j)) =
                  (u,
                   REST(v),
                   AMB(STRICTIFY(u, TRUE),
                       STRICTIFY(FIRST(REST(v)), FALSE),
                   i) where i ∈ ω.
```

Notation: As before with I' for I:

$I'[\![<x_1\ x_2\ \ldots\ x_k>]\!] = I'[\![fons_T(x_1, <x_2\ \ldots\ x_k>)]\!]$;

$I'[\![\{x_1\ x_2\ \ldots\ x_k\}]\!] = I'[\![fons_F(x_1, \{x_2\ \ldots\ x_k\})]\!]$;

$I'[\![\{x_1|\ x_2\ \ldots\ x_k\}]\!] = I'[\![fons_T(x_1, \{x_2\ \ldots\ x_k\})]\!]$;

$I'[\![<>]\!] = NIL = I'[\![\{\}]\!]$.

The interpretation of "structure expressions" is a quadruple, but that quadruple is unrelated to the triple which is a fern; it is a nested structure quite similar to the direct union which McCarthy [39] used to interpret sequences. There is a natural interpretation here also of the value β in $\underline{fons}_\beta$; if it is T then the third item in the interpretation will be TRUE and if it is F then AMB introduces some uncertainty. In the following theorems we shall see that this uncertainty is resolved by convergence, much like the way that edges are added to ferns by EUREKA under interpretation I.

In both the fern semantics and the structure semantics defined here we note that the interpretations of the arguments to $\underline{fons}_\beta$ are embedded within the result. In the case of ferns, those interpretations became part of the labelling; here the interpretations are embedded more directly into the quadruple (and sometimes under AMB in the third position). It is necessary that the triple which is the fern--or quadruple which is the structure-- be constructed with these interpretations suspended (called-by-delayed-value) until they are needed; otherwise these definitions are futile. Most importantly, the invocation of AMB is postponed until either first or rest of a structure is to be interpreted and then that invocation in that structure never occurs again!

<u>Definition</u>: (equivalence)

$I[\![f]\!] \simeq I'[\![f]\!]$ iff

$I[\![f]\!] = I'[\![f]\!]$ or

($I[\![first(f)]\!] \simeq I'[\![first(f)]\!]$ and

$I[\![rest(f)]\!] \simeq I'[\![rest(f)]\!]$).

<u>Lemma 1</u>: If $I[\![x]\!] = I'[\![x]\!]$ whenever x is an elementary expression (i.e. not involving structures), and if f is an expression of the form $fons_F(x_1, fons_F(x_2, \ldots fons_F(x_{n-1}, fons_F(x_n, y)) \ldots))$ where $n = \omega$ or $y = NIL$ (corresponding to the notation $\{x_1\ x_2 \ldots x_n\}$) where every x_i $(i > 0)$ is an elementary expression and at <u>most</u> one x_j is such that $I[\![x_j]\!] \neq \bot$, then $I[\![f]\!] \simeq I'[\![f]\!]$.

<u>Proof</u>: If $n = 0$ then $I[\![f]\!] = NIL = I'[\![f]\!]$. If $n > 0$ and all $I[\![x_i]\!] = \bot$ then all $I'[\![x_i]\!] = \bot$ because each x_i is elementary. Hence EUREKA cannot converge so $I[\![first(f)]\!] = I[\![rest(f)]\!] = \bot$. Moreover, all the STRICTIFY factors to AMB must similarly diverge since they are strict in each $I'[\![x_i]\!]$ so $I'[\![first(f)]\!] = I'[\![rest(f)]\!] = \bot$. If $n > 0$ and there is a j such that $I[\![x_j]\!] = a \neq \bot$ then $I'[\![x_j]\!] = a$ and $I[\![x_i]\!] = \bot = I'[\![x_i]\!]$ for $i \neq j$. The strictness properties of EUREKA require that EUREKA($I[\![f]\!]$) make the node associated with x_j minimal. Similarly, the STRICTIFY patterns require that the only TRUE values in the structure of $I'[\![f]\!]$ be associated with $I[\![x_j]\!]$. So $I[\![first(f)]\!] = I[\![x_j]\!] = I'[\![first(f)]\!]$. If $n = 1$ then $I[\![rest(f)]\!] = NIL = I'[\![rest(f)]\!]$; otherwise $I[\![rest(f)]\!] \simeq I'[\![rest(f)]\!]$ by the argument preceding when all $I[\![x_i]\!] = \bot$. □

Lemma 2: If $I[\![x]\!] = I'[\![x]\!]$ whenever x is an elementary expression and if f is an expression of the form $fons_F(x_1, fons_F(x_2, \ldots fons_T(x_n,y) \ldots))$ (corresponding to $\{x_1\ x_2 \ldots x_n| \ldots \}$) where $I[\![y]\!] \simeq I'[\![y]\!]$, x_i $(i > 0)$ is an elementary expression and at most one x_j is such that $I[\![x_j]\!] \neq \bot$, then $I[\![f]\!] \simeq I'[\![f]\!]$.

Proof: The proof above suffices except when $n = 1$ or $j = n$. When $n = 1$, then $I[\![first(f)]\!] = I[\![x_n]\!]$ regardless of whether it is $\bot$ or not because EUREKA has no choice to make; $I[\![rest(f)]\!] = I[\![y]\!]$ then. In that case $I'[\![first(f)]\!] = I'[\![x_n]\!]$ since there is no STRICTIFY to resolve and $I'[\![rest(f)]\!] = I'[\![y]\!]$. So $I[\![f]\!] \simeq I'[\![f]\!]$. If $n > 1$ and $j = n$ then $I[\![first(f)]\!] = I[\![x_n]\!] = I'[\![x_n]\!] = I'[\![first(f)]\!]$ for reasons similar to the proof of Lemma 1, but $I[\![rest(f)]\!]$ is $I[\![y]\!]$ with n-1 leaves labelled $\bot$ added. Similarly, $I'[\![rest(f)]\!]$ is a structure in which the STRICTIFY pattern defers to convergence in $I'[\![y]\!]$. It turns out that

$$I[\![fons_F(x_1, fons_F(x_2, \ldots fons_F(x_{n-1},y) \ldots))]\!] \simeq I'[\![fons_F(x_1, fons_F(x_2, \ldots fons_F(x_{n-1},y) \ldots))]\!]$$

where $I[\![x_i]\!] = \bot$ so $I[\![rest(f)]\!] = I'[\![rest(f)]\!]$. □

Definition: (similarity)

$I[\![f]\!] \sim I'[\![f]\!]$ iff

$I[\![f]\!] = I'[\![f]\!]$ or the sets of possible interpretations

$P = I[\![<first(f)\ rest(f)>]\!]$; (P is a subset of the range of I)

$P' = I'[\![<first(f)\ rest(f)>]\!]$; (P' is a subset of the range of I')

(Where I and I' may be relations rather than functions.)

are such that $\forall (x,r) \in P\ \exists (x',r') \in P'$ such that $x \sim x'$ and $r \sim r'$ and $\forall (x',r') \in P'\ \exists (x,r) \in P$ such that $x \sim x'$ and $r \sim r'$.

Lemma 3 If $I[\![f]\!] \simeq I'[\![f]\!]$ then $I[\![f]\!] \sim I'[\![f]\!]$.

Proof: Under the $\simeq$ definition there is little choice for interpretation because equality at the eventual elementary labels is necessary. Thus, P and P' are singleton sets and so $I[\![f]\!] \sim I'[\![f]\!]$ trivially.

Lemma 4: If $I[\![x_i]\!] = I'[\![x_i]\!]$ for $1 \le i \le n$ and f is an expression of the form $fons_F(x_1, fons_F(x_2, \ldots fons_F(x_n,y) \ldots))$ where $n = \omega$ or $y = NIL$ then $I[\![f]\!] \sim I'[\![f]\!]$.

Proof: The argument follows that of Lemma 1 but the order of elements may be permuted if there is more than one path that EUREKA or STRICTIFY--with AMB-- can choose. For any $I[\![x_j]\!] \ne \bot$, however, we can envision a quantum of time when $I[\![x_i]\!] = \bot$ for $i \ne j$, apply Lemma 1 and then allow $I[\![x_i]\!]$ to be whatever value is necessary. This trick may be applied successively to $rest^k(f)$; using oracles to select x_j in various orders. For each choice we may envision various elements of P and P' respectively, according to the order that the non-$\bot$ elements were chosen. The sets P, P' generated this way, have associated pairs (x,r) and (x',r') added by applying similar oracles. This generation includes all interpretations when all oracles are considered. Hence $I[\![f]\!] \sim I'[\![f]\!]$. □

Lemma 5: If $I[\![x_i]\!] = I'[\![x_i]\!]$ for $1 \le i \le n$ and f is an expression of the form $fons_F(x_1, fons_F(x_2, \ldots fons_T(x_n,y) \ldots))$ where $I[\![y]\!] \sim I'[\![y]\!]$ then $I[\![f]\!] \sim I'[\![f]\!]$.

Proof: The argument follows that of Lemma 2 using Lemma 4. □

<u>Theorem</u>: If $I[\![x]\!] = I'[\![x]\!]$ when x is an elementary (non-fern) expression then for all expressions x, $I[\![x]\!] \sim I'[\![x]\!]$.

<u>Outline of Proof</u>: We must also include expressions involving <u>null</u> which were not considered in the lemmas, but <u>null</u> is never strict in the interpretations <u>within</u> a fern so we may consider it as an elementary function. When x is a fern-expression we need only apply <u>Lemma 4</u> or <u>5</u> if it yields a fern of elementary values. If the fern has "depth" or subferns, then <u>Lemmas 4</u> and <u>5</u> must be extended; this can be done recursively with the pairs in P, P' having depth along the first projection (x and x' in the definition of ~) as well as the second (r and r' as already occurs with <u>Lemmas 4</u> and <u>5</u>) ∎

The theorem states that for any interpretation under either fern or structure semantics there is an interpretation under the other. We cannot require that the semantics of fern expressions be the <u>same</u> when nondeterminism (EUREKA or AMB) is involved. We can, however, say that a given meaning is <u>possible</u> in the sense that the program would allow it to be chosen as well. Thus the semantics are "sufficiently" similar to be the same. In any two implementations the way that nondeterministic choices are made will determine whether we can say that they are the same in the strong (≃) sense. Here we are satisfied with a weak (~) similarity.

Conclusion

We have presented a single structure builder for applicative languages known as *fons*. The semantics of *fons* are given in two different, but related formulations. The first, a restriction of a forest called a *fern*, offers the user a visualization of the data structures which result from using *fons*; it also offers insight into the terminology which we have coined and which may be comfortably used when dealing with the other semantics. The second semantics is more abstract but also more concise. Formalists and implementors should prefer it to the first.

These two semantics are shown to be equivalent in a weak sense: that any result under one interpretation is also possible under the other. The argument does not approach the formality of similar powerdomain constructions [45,51], but it is sufficient for the purposes here.

We have been careful to embed the interpretation of any element *fons*ed into a fern *into* that fern. The subsequent interpretations of *first* and *rest* on that fern will always recover the original interpretation of that element regardless of whether or not the interpreting function has changed in the meantime (say, by creating a new environment of lambda bindings.) That embedded interpretation need not be completed until later when the structure is probed; it may be suspended until then.

Indeed, the embedded interpretations play exactly the same role within a data structure as the suspensions play within list structures (here called sequences). The suspension of content within data structures is necessary for implementing call-by-need semantics in a structure language, and such an implementation is sufficient to impose call-by-need semantics when the structures so suspended are the environments.

Here again we are using call-by-need semantics when we prescribe the _a posteriori_ effects of EUREKA in a fern or the unique invocation of each AMB expression in a structure. If we have two references to the _same_ structure then the probings of that structure are always the same. Repeatedly probing the _same_ structure _must_ give the same results. Although there is choice to be made in ordering the fern, that choice is made but once. If we attempt to _build_ the same fern twice, however, then two ferns result and the strongest claim we can make is, as in the theorem, similarity. Strong semantic equivalence (≃) is available on a data structure built _once_, but if it is rebuilt, performance is not the same--only similar (~). The "onceness" perspective does not allow, however, for structures to be shared.

We have chosen to embed nondeterminism in structure in order to take advantage of the programmer's existing respect for data structures. Because of space conservation goals, most users already try to share structures as much as possible; such habits preclude problems of foreseeably similar ferns (because if the similarity is predictable then the second fern should be a borrowed

reference to the first). Thus, practice--not semantics-- will advance semantic equivalence of structures.

There is a perspective on ferns which bears mention. We specified formal semantics as labelled ferns in a way in which the indeterminate property (the choice of edges to be added) is separated from the semantics of content (the labels themselves). One useful interpretation of the semantics of the fern is that the edges are not added to yield a topological sort, but that they were there all the time in the guise of an oracle which would add the correct edges. All the effort given here merely prescribes the correct oracle. From that perspective we are rather close to the powerdomain arguments.

Thus the single constructor fons is handed to the user in two flavors: cons and frons. An interpreter for an earlier version of cons/frons exists [30] and work continues on a newer version. This style of programming is being applied to the real time problems of patient monitoring [50] and radar tracking [49]. Another paper [21] addresses the issue of fairness in resolving indeterminant choices.

Acknowledgement: We thank Steven D. Johnson and particularly Mitchell Wand for numerous constructive discussions and critical readings. Early encouragement from John Backus and James H. Morris, Jr., pushed this work along. We also thank Carl Hewitt who suggested the airline reservation problem and Robert Tennent who offered some thoughtful proposals on an earlier version.

REFERENCES

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21, 8 (August, 1978), 613-641.

2. Berkling, K. J. Reduction languages for reduction machines. Second Annual Meeting of Computer Architecture (1975), 133-138.

3. Brinch Hansen, P. Operating System Principles, Prentice-Hall, Englewood Cliffs, NJ (1973).

4. Burge, W. H. Recursive Programming Techniques, Addison-Wesley, Reading, MA (1975).

5. Burstall, R. M., and Darlington, J. A transformation system for developing recursive programs. J. Assoc. Comput. Mach. 22, 1 (January, 1975), 129-144.

6. Church, A. The Calculi of Lambda Conversion (Ann. of Math. Studies 6), Princeton Univ. Press, Princeton (1941).

7. Curry, H. B., and Feys, R. Combinatory Logic I, North-Holland, Amsterdam (1958).

8. Dennis, J. B. First version of a data flow language. In B. Robinet (ed.), Programming Symposium, Springer, Berlin (1974), 362-376.

9. Dennis, J. B. A language design for structured concurrency. In J. H. Williams and D. A. Fisher (eds.), Design and Implementation of Programming Languages, Springer, Berlin (1977), 231-242.

10. Dijkstra, E. W. Co-operating sequential processes. In F. Genuys (ed.), Programming Languages, Academic Press, London (1968), 43-112.

11. Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E.F.M. On-the-fly garbage collection: an exercise in cooperation. Comm. ACM 21, 11 (November, 1978), 966-975.

12. Donovan, J. J., and Madnick, S. E. Software Projects, McGraw-Hill, New York (1977), 377-383.

13. Floyd, R. W. Nondeterministic algorithms. J. Assoc. Comput. Mach. 14, 4 (October, 1967), 636-644.

27. Hewitt, C. E., and Atkinson, R. Parallelism and synchronization in actor systems. Proc. 4th ACM Symp. on Principles of Programming Languages (1977), 267-280.

28. Hewitt, C. E., and Smith, B. Towards a programming apprentice. IEEE Trans. Software Engrg. SE-1, 1 (March, 1975), 26-45.

29. Hoare, C.A.R. Monitors: an operating system structuring concept. Comm. ACM 17, 10 (October, 1974), 549-557.

30. Johnson, S. D. An Interpretive Model for a Language Based on Suspended Construction. M.S. thesis, Indiana University (1977).

31. Kahn, G., and MacQueen, D. Coroutines and networks of parallel processes. In B. Gilchrist (ed.), Information Processing 77, North-Holland, Amsterdam (1977), 993-998.

32. Keller, R. M. Denotational models for parallel programs with indeterminant operators. In E. J. Neuhold (ed.), Formal Description of Programming Concepts, North-Holland, Amsterdam (1978), 337-366.

33. Knuth, D. E. The Art of Computer Programming 1, Fundamental Algorithms (2nd ed.), Addison-Wesley, Reading, MA (1973).

34. Knuth, D. E. The Art of Computer Programming 2, Seminumerical Algorithms, Addison-Wesley, Reading, MA (1969), 551.

35. Kosinski, P. R. A data flow language for operating systems programming. Proc. ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (September, 1973), 89-94.

36. Lamport, L. Proving the correctness of multiprocess programs. IEEE Trans. on Software Engineering SE-3, 2 (March, 1977), 125-143.

37. Landin, P. J. A correspondence between ALGOL 60 and Church's lambda notation. Comm. ACM 8, 2 (February, 1965), 89-101.

38. Landin. P. J. The next 700 programming languages. Comm. ACM 9, 3 (March, 1966), 157-162.

39. McCarthy, J. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (eds.), Computer Programming and Formal Systems, North-Holland, Amsterdam (1963), 33-70.

40. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. E. LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA (1962), Chapter 1.

41. Manna, Z. Mathematical Theory of Computation, McGraw-Hill, New York (1974), Chapter 5.

42. Manna, Z., and McCarthy, J. Properties of programs and partial functional logic. In B. Meltzer and D. Michie (eds.), Machine Intelligence 5, Edinburgh Univ. Press, Edinburgh (1970), 27-37.

43. d'Onfrio, C. Le Fontane di Roma, Staderini, Rome (1957), 22.

44. Owicki, S., and Gries, D. An axiomatic proof technique for parallel programs I. Acta Informatica 6, 4 (August, 1976), 319-340.

45. Plotkin, G. D. A powerdomain construction. SIAM J. Comput. 5, 3 (September, 1976), 452-487.

46. Rabin, M.O., and Scott, D. S. Finite automata and their decision problems. IBM J. Res. Develop. 3, 2 (April, 1959), 114-125. Also in E. F. Moore (ed.), Sequential Machines, Addison-Wesley, Reading, MA (1964), 63-91.

47. Robinson, L., and Levitt, K. N. Proof techniques for hierarchically structured programs. Comm. ACM 20, 4 (April, 1977), 271-282.

48. Scott, D. S. Logic and programming languages. Comm. ACM 20, 9 (September, 1977), 634-641.

49. Smoliar, S. W. Using applicative techniques to design distributed systems. Proc. Specifications of Reliable Software, IEEE Cat. No. 79 CH1401-9C (April, 1979)., 150-161.

50. Smoliar, S. W., and Palmer, D. F. An applicative model for the design and analysis of distributed architectures. Internal Working Paper, General Research Corp., Santa Barbara, CA.

51. Smyth, M. B. Power domains. J. Comp. Sys. Sci. 16 (1978), 23-36.

52. Steele, G. L., Jr. LAMBDA: the ultimate declarative. A.I. Memo No. 379, Mass. Inst. of Tech., Cambridge (1976).

53. Steele, G. L., Jr. RABBIT: a compiler for SCHEME. A.I. Tech. Rept. No. 474, Mass. Inst. of Tech., Cambridge (May, 1978).

54. Vuillemin, J. Correct and optimal implementation of recursion in a simple programming language. J. Comp. Sys. Sci. 9, 3 (June, 1974), 332-354.

55. Wadsworth, C. Semantics and Pragmatics of Lambda-calculus, Ph.D. dissertation, Oxford (1971).

56. Waldinger, R. J., and Levitt, K. N. Reasoning about programs. Artificial Intelligence 5, 3 (Fall, 1974), 235-316.

57. Wand, M. Continuation-based program transformation strategies. J. Assoc. Comput. Mach. (to appear).

58. Wegbreit, B. Goal-directed program transformation. IEEE Trans. Software Engrg. SE-2, 2 (June, 1976), 69-79.

59. Yonezawa, A., and Hewitt, C. E. Modelling distributed systems. 5th Intl. Joint Conf. on Artificial Intelligence (1977), 370-377.

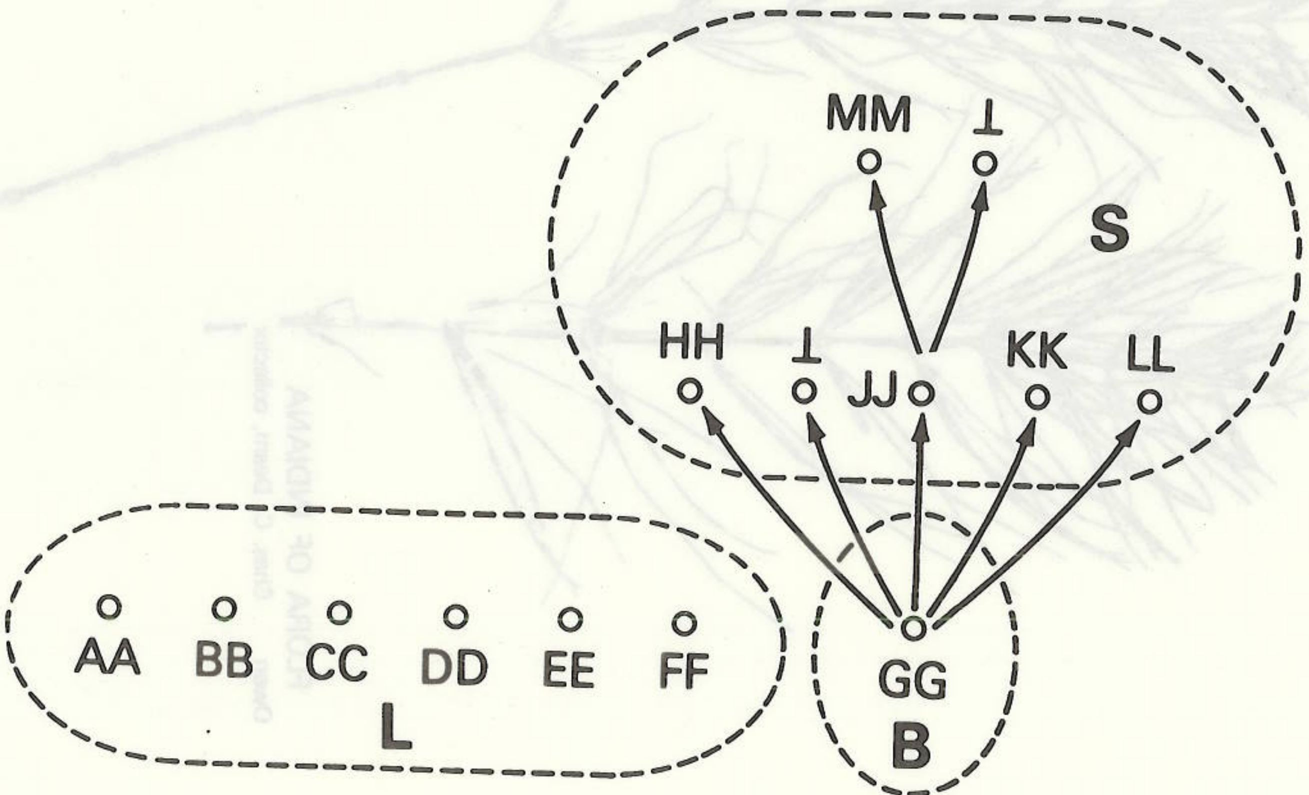

Figure 1. A labelled fern, I⟦ f ⟧ .

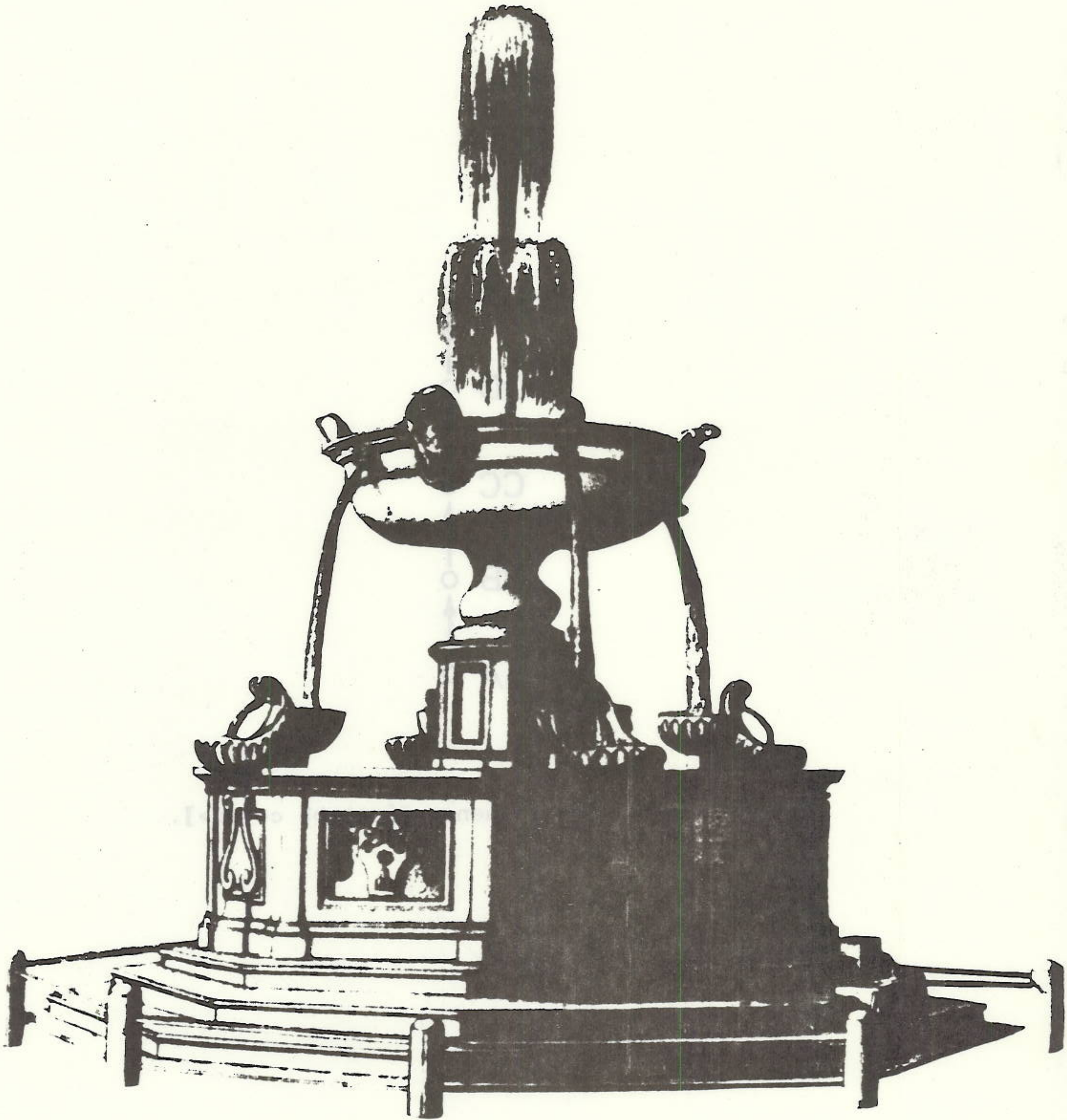

Figure 3. Fons olei (also known as La fontana di S. Maria in Trastevere [43]) in 1675.

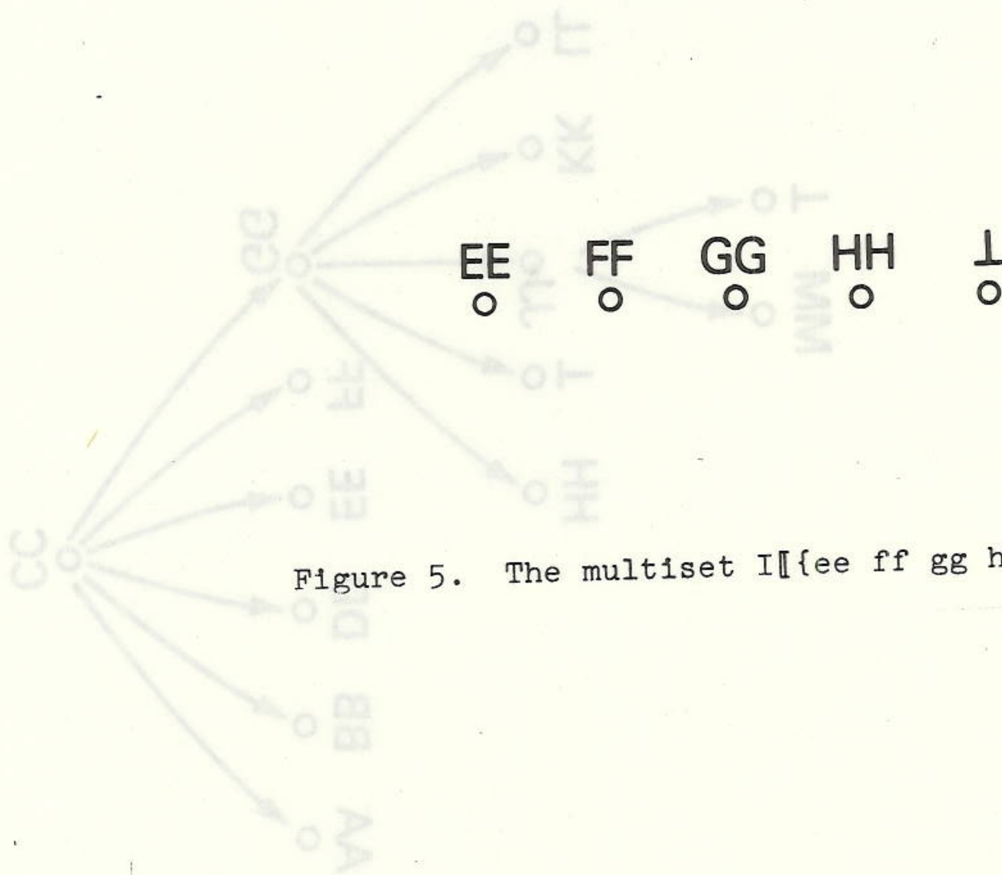

Figure 5. The multiset I⟦{ee ff gg hh}⟧.

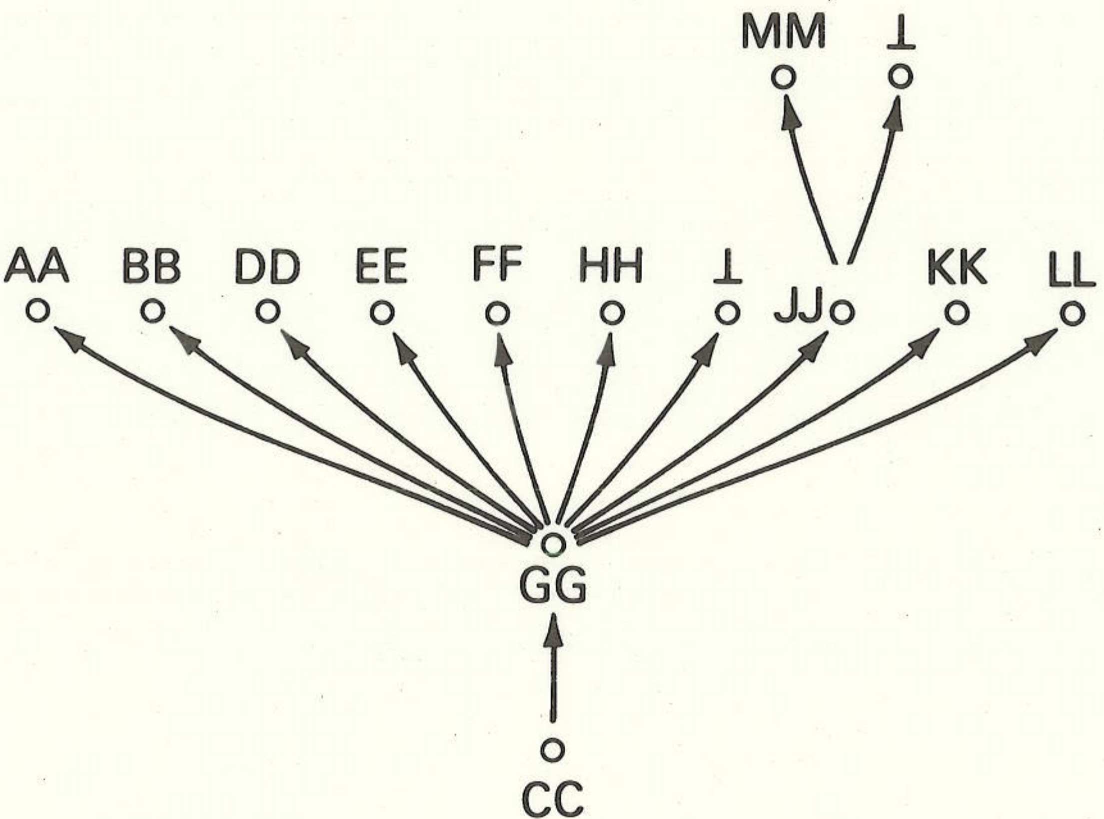

Figure 7. I⟦ first:rest:f ⟧ = GG .