Fancy Ferns Require Little Care*
(Extended Abstract)
by

Daniel P. Friedman and David S. Wise†

Abstract: A fern is a kind of tree with at most one non-terminal node at every level. It is defined as a structure for encapsulating results of asynchronous, indeterminate multiprocessing. Formal properties of two structure builders, CONS and FRONS, are proved; most interesting are the idempotence properties regarding FRONS and ⊥, and the contrasting order-preserving behavior of CONS. We present a simple language using these operations through axioms and examples. Finally, practical demands of shared references, programming practice, and fairness are demonstrated to refine the axiomatized language. This separation of reference-sharing issues from the definitions on indeterministic choice, and its subsequent derivation, are important steps toward formal semantics for indeterminism.

Keywords and phrases: asynchronous recursion, fern, amb, multiset, bag, frons, cons, list, prefix*, stream, indeterminism, call-by-need, parallel processing, suspension, lazy evaluation, unpredictable ordering, arbiter, arbit.

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

---

*Headline The New York Times Sunday, Apr. 29, 1979, page D39 vol. 128 , issue 44,202.

†Research supported in part by The National Science Foundation under grant numbers MCS77-22325 and MCS79-04183.

Introduction

This paper presents a formal development of ferns and integrates them into the environment of applicative programming practice. We have developed the operational semantics of ferns elsewhere [6, 5] to a degree far in advance of the theoretical thrust of this paper. Here we only put that work on as firm a footing as possible.

Ferns are defined as a data structure which may first be perceived as a generalization of list structures (a la LISP[12]) designed to include an (initially) unordered structure here dubbed a multiset. Our perception of list is firmly rooted in an applicative-style programming, that precludes side-effects (assignment imperatives). We derive our generalization from the rule that a list's content, once the list is defined, is immutable. Since content need not even be evaluated at the time a list is constructed [2], a list built as a totally ordered object may include values represented as "suspensions," immutable information sufficient to generate those values as in call-by-need [21,20,7] semantics for parameter passing. Because these suspensions are analogous to dormant processes, parallelism can be introduced thereby [3].

A multiset [9] (also bag [22]) is essentially a list built without specifying explicit order. As we shall see in the last section of this paper, one can argue that order can be impressed upon the structure during its use, but for now we shall allow its order to remain unspecified except for one convention. As an unordered structure, its access is not deterministic and involves choice. We constrain that choice through mechanisms to be associated with EUREKA images or ε-abstraction, so that alternatives involving suspensions which are ill-behaved are avoided. In terms of denotational semantics [17], we will only allow choices associated with non-$\bot$ values (or with ordering forced at construction time - i.e., list orderings).

In introducing this convention, which we associate with the term "indeterminism", we argue briefly that call-by-need is a natural evaluation strategy for operational semantics. A critical ability of computers (aside from making decisions) is the ability to recall rapidly values already known. Typically this is done by fetching the value of a variable from memory; in lambda-calculus the analog is recovering the bound value of a variable. In lambda-calculus models various protocols are available to effect this during the evaluation of an expression: call-by-name, call-by-value, call-by-need; all have more or less the same effect in terms of Church-Rosser equivalence. The last two, however, are operationally closest to the "table-look-up" behavior of computer memory efficient in time since no argument is evaluated twice. As we argue next, this avoidance of "twiceness" is necessary to grapple with indeterminism.

Indeterminism is a facility often used by a programmer to incorporate external behavior into his program through selection of "good" values and avoidance of "bad" ones. Such a faciltiy specifies a choice which is not determined just by program or by input values, but may involve time dependencies among various input streams. (It is not to be confused with the non-determinism

of automata theory, a stylistic convenience for compressing explosive deterministic computations.) A program which uses indeterminism, having made an underspecified choice may refer repeatedly to the results of that choice.

We present a formal definition of ferns for representing indeterminism. The first section includes some notation, definitions, and examples of ferns -- the generalization of lists but built from multisets. The definitions of CONS - the list builder, FRONS - the multiset-builder, and EUREKA - the choice builder provide for building and probing ferns. Formal results at this level establish the mathematical properties of these definitions that later support the language built on them. A congruence relation is established on the set of ferns and forms the basis for several important results: that FRONS is insensitive to $\bot$ (Corollary 6.3 on idempotency), that FRONS avoids only $\bot$ (Corollary 7.1), and in contrast that CONS preserves order regardless of whether or not its domain argument is $\bot$ (Theorem 8). Infinite ferns are also considered. Theorem 10, establishing that any choice available in decomposing an (underordered) fern is also available if that fern were CONSed in the first place, forms the foundation of the later language definition: that unordered ferns may be accessed as if they had been constructed specifying some total order (as CONS does).

This supports the semantics presented in the following section, which provides that a fern appears to be any one of its CONS ∘ EUREKA images (where ∘ denotes composition), the choice of image being made on every application of the ε-abstraction. There the user is provided a mutating frons, <u>μfrons</u>, constructor in addition to McCarthy's <u>cons</u> [12]. Examples there include <u>amb</u>, <u>arbit</u>, symmetric <u>or</u>, and <u>merge</u> of arbitrary numbers of streams. In the final section we justify the restriction of <u>μfrons</u> to a non-mutating <u>frons</u>, available without changing any axioms. As a result, an indeterministic choice need only be made once within the language and preserved within a fern structure.

<u>Conventional Notation</u>

A sequence over an alphabet is any string composed of symbols from the alphabet. The empty sequence is Λ. If V is an alphabet, then V^* denotes the set of all finite sequences over V, including Λ. V^ω denotes the set of all infinite sequences over V. A sequence S may be perceived as a partial function from the natural numbers, ω, to the alphabet. Thus we might write $S \in V^*$ as $S = s_0 \cdot s_1 \cdot s_2$ for $s_i \in V$. Because there are symbols in our alphabet which are not represented by a single character we require the syntactic use of a concatenation symbol "·" between symbols in a sequence. Some alphabets include sets as elements.

We also use the conventional powerset operator: $PS = \{X \mid X \subseteq S\}$ and function extension operator: $f[y/x] = \lambda u.(u{=}x \rightarrow y,\ fu)$ where the (→,) form is the strict conditional. Finally, we will be referring to an arbitrary Scott Domain, D, and its bottom element (⊥).

A <u>multiset</u> over D is a mapping, $M: D \rightarrow \omega{+}1$. We interpret the multiset as a function mapping $d \in D$ into the number of times (perhaps infinite) that it occurs in the multiset. The set of multisets over D is $M(D)$ abbreviated to M where obvious. Singleton multisets are written using set notation; if $d \in D$ then $\{d\} = (\lambda x.x{=}d \rightarrow 1,\ 0) \in M(D)$. We define additive operators on multisets analogous to set union and difference [9 22].

If $M,N \in M(D)$ then

$$M \oplus N = \lambda x.M(x) + N(x);$$

$$M \ominus N = \lambda x.M(x) \dot{-} N(x).$$

Definition: The set of ferns over D is

$$F(D) = \{A_0 \cdot a_1 \cdot A_2 \cdot a_3 \cdot A_4 \cdot \ldots \in (M(D) \cdot D)^* \cdot M(D) \cup (M(D) \cdot D)^\infty \mid \forall i \geq 0 \; A_{2i}(a_{2i+1}) > 0\}.$$

We write an element of $F(D)$ in Old English font: $\mathfrak{A} \in F(D)$. If $\mathfrak{A} \in F(D)$ then we write $\mathfrak{A} = A_0 \cdot a_1 \cdot A_2 \cdot a_3 \cdot \ldots$ where $A_{2i} \in M(D)$ and $a_{2i+1} \in D$. This indexing is taken to be conventional so that the meaning of A_{2i} and a_{2i+1} are all notationally implied once $\mathfrak{A}$ is known. Similarly B_0 and b_1 (C_0 and c_1) when $\mathfrak{B}$ (respectively $\mathfrak{C}$) is defined.

A trivial element of F is the empty fern $\mathfrak{Nil} = \lambda x.0$.

Example of a simple fern

We next present a simple example of three related ferns. That these ferns are built from one another reflects the operational nature of the sharing inherent in an implementation.

The choice of the word "fern" (and later of "FRONS") is motivated by the shape of these structures as illustrated (Figure 1) in the following examples. The reader is encouraged to compare those figures with that of equisetum in Figure 2, which is not quite a tree.

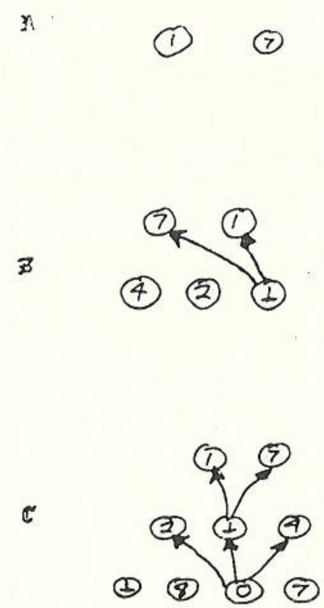

Figure 1. The example ferns $\mathfrak{A}$, $\mathfrak{B}$, and $\mathfrak{C}$.

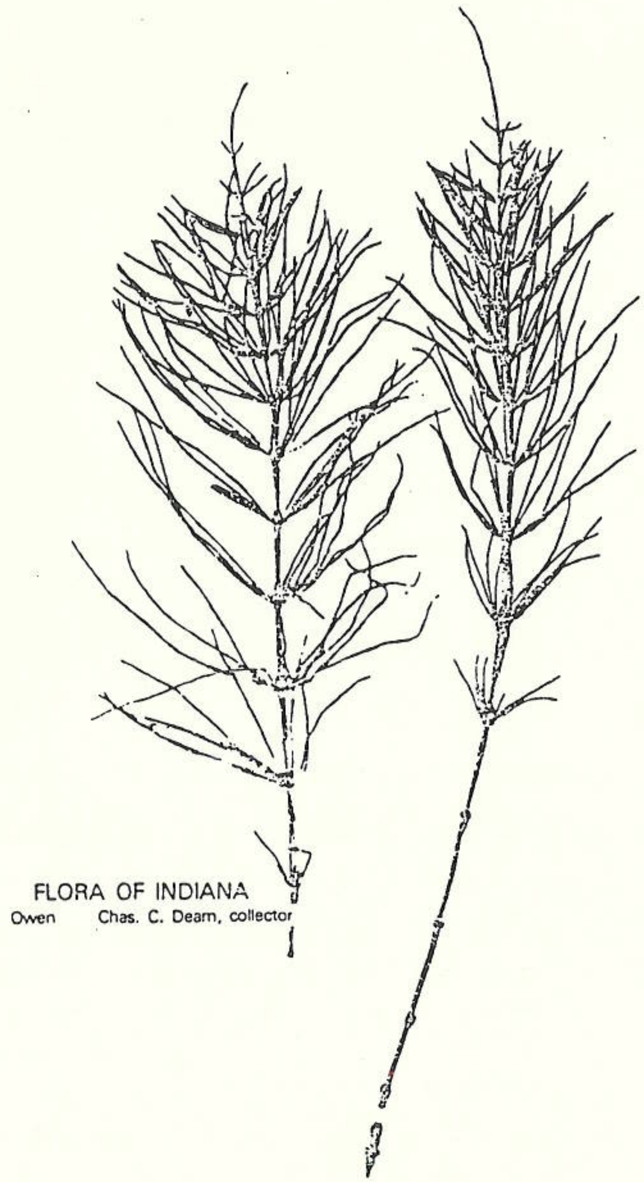

Figure Z. Equisetum arvense (Courtesy of Herbarium of Indiana University).

$\mathfrak{A} = A_0$

$A_0 = \{1, 7\}$

$\mathfrak{B} = B_0 \cdot b_1 \cdot B_2$

$B_0 = \{2, 4, \perp\}$

$b_1 = \perp$

$B_2 = A_0$

$\mathfrak{C} = C_0 \cdot c_1 \cdot C_2 \cdot c_3 \cdot C_4$

$C_0 = \{\perp, 8, 0, 7\}$

$c_1 = 0$

$C_2 = B_0$

$c_3 = b_1$

$C_4 = B_2 = A_0$

A <u>list</u> is a fern, $K \in F$, such that K_{2i} is a singleton or $K_{2i} = \lambda x.0$ for all i. Note that a list is composed of singleton multisets. For example, the LISP [12] list (1 2 1 3) is represented by $K = \{1\} \cdot 1 \cdot \{2\} \cdot 2 \cdot \{1\} \cdot 1 \cdot \{3\} \cdot 3 \cdot \lambda x.0$.

The purpose of CONS and FRONS is to build up ferns as similar constructors build up lists in LISP [12]. The function EUREKA specifies a set of possible pairs which specify how a fern is to be probed in a "bottom-avoiding" fashion. It is reminiscent of other <u>choice</u> functions: <u>amb</u> [12], <u>arbit</u>[8], and <u>merge</u>[1] which avoid results of ill-defined or misbehaved computation, but we use it only descriptively: it is not part of a user's language.

Notation: For the following three definitions we specify a local string notation on ϕ and ψ. Define ϕ and ψ by $\mathbb{A} = A_0 \cdot \phi$ and, if $\phi \neq \Lambda$ then $\phi = a_1 \cdot A_2 \cdot \psi$. This notation references the suffices after the first one or three items in $\mathbb{A}$.

CONS: $D \times F \to F$

$(d,\mathbb{A}) \mapsto \{d\} \cdot d \cdot \mathbb{A}$

FRONS: $D \times F \to F$

$(d,\mathbb{A}) \mapsto \{d\} \oplus A_0 \cdot \phi$

EUREKA: $F \to P(D \times F)$

$\mathbb{A} \mapsto \{<a_1,(A_2 \oplus A_0 \ominus \{a_1\}) \cdot \psi>|\ A_0=\{a_1\} \vee a_1 \neq \iota\}$
$\cup \{<a,(A_0 \ominus \{a\}) \cdot \phi>|\ a \neq \iota \ \&\ A_0(a)>0 \ \&\ (a=a_1) \supset (A_0(a)>1)\ \}$

One way of forming our example fern is by generating $\mathbb{B}$ from $\mathbb{A}$ and then $\mathbb{C}$ from $\mathbb{B}$:

$\mathbb{A}$ = FRONS(1,FRONS(7,Nil))

$\mathbb{B}$ = FRONS(2,FRONS(4,CONS(ι,$\mathbb{A}$)))

$\mathbb{C}$ = FRONS(8,FRONS(ι,FRONS(7,CONS(0,$\mathbb{B}$)))).

EUREKA for ferns $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$ follows:

EUREKA($\mathbb{A}$) = {<1,{7}>, <7,{1}> }

EUREKA($\mathbb{B}$) = {<2,{ι,4}$\cdot\iota\cdot\mathbb{A}$>, <4,{$\iota$,2}$\cdot\iota\cdot\mathbb{A}$> }

EUREKA($\mathbb{C}$) = {<0,{ι,ι,2,4,8,7}$\cdot\iota\cdot\mathbb{A}$>,
<7,{ι,0,8}$\cdot 0\cdot\mathbb{B}$>, <8,{ι,0,7}$\cdot 0\cdot\mathbb{B}$> }.

Theorem 1: FRONS(d,Nil) = {d}; CONS(d,Nil) = {d}$\cdot$d$\cdot$(λx.0).

Definition: If $\mathbb{A} \in F$ then a promotion sequence of F is a sequence $\{<d_i,\mathbb{A}_i>\}_{i>0} \in (D\times F)^* \cup (D\times F)^\infty$ such that $\mathbb{A}_0 = \mathbb{A}$ and $<d_{i+1},\mathbb{A}_{i+1}> \in$ EUREKA($\mathbb{A}_i$) for $i \geq 0$. A promotion sequence of length n is maximal if either $n < \omega$ and EUREKA($\mathbb{A}_n$) = $\emptyset$ or $n = \omega$.

There are infinite ferns whose maximal promotion sequences are all finite: for example, $\lambda x.\iota$ whose triviality anticipates Corollary 3.2 and Corollary 6.3.

One (of several) promotion sequences of $\mathbb{B}$ is:
$<4,\mathbb{B}_1>$, $<2,\mathbb{B}_2>$, $<\iota,\mathbb{B}_3>$, $<7,\mathbb{B}_4>$, $<1,\mathbb{B}_5>$ where

$\mathbb{B}_1$ = {ι,2}$\cdot\iota\cdot\mathbb{A}$

$\mathbb{B}_2$ = {ι}$\cdot\mathbb{A}$

$\mathbb{B}_3$ = $\mathbb{A}$;

$\mathbb{B}_4$ = {1}

$\mathbb{B}_5$ = Nil.

Definition: We define the binary relation < on F by $\mathbb{A} < \mathbb{B}$ when for every maximal promotion sequence $\{<d_i,\mathbb{A}_i>\}_{i>0}$ (of $\mathbb{A}$) there is a promotion sequence $\{<d_i,\mathbb{B}_i>\}_{i>0}$ (of $\mathbb{B}$).

The promotion sequence of $\mathbb{B}$ need not be maximal, but may be longer than that of $\mathbb{A}$ so long as the $\{d_i\}$ coincide for the length of $\mathbb{A}$'s maximal promotion sequence.

<u>Theorem 2</u>: $\mathcal{A} < \mathcal{B}$ if and only if for every $\langle d, \mathcal{A}' \rangle \in$ EUREKA($\mathcal{A}$) there exists $\langle d, \mathcal{B}' \rangle \in$ EUREKA($\mathcal{B}$) such that $\mathcal{A}' < \mathcal{B}'$.

<u>Corollary 2.1</u>: The $<$ relation is reflexive and transitive.

<u>Definition</u>: The binary relation $\approx$ on F is defined by $\mathcal{A} \approx \mathcal{B}$ if and only if $\mathcal{A} < \mathcal{B}$ and $\mathcal{B} < \mathcal{A}$.

<u>Corollary 3.1</u>: The $\approx$ relation is an equivalence relation on F.

Using the fern $\mathcal{B} = B_0 \cdot b_1 \cdot B_2$ from our earlier example, form $\mathcal{B}' =$ FRONS($\bot, \mathcal{B}$) $= \{\bot\} \oplus B_0 \cdot b_1 \cdot B_2$. The additional $\bot$ in B_0' interferes with the construction of maximal promotion sequences for $\mathcal{B}'$ equal in length to those for $\mathcal{B}$. In particular for each promotion sequence $\{\langle d_i', \mathcal{B}_i' \rangle\}_{i>0}$ for $\mathcal{B}'$ there is no $d_i' = \bot$; whereas $d_3 = \bot$ in every maximal promotion sequence $\{\langle d_i, \mathcal{B}_i \rangle\}_{i>0}$ for $\mathcal{B}$. Hence $\mathcal{B}' < \mathcal{B}$ but $\mathcal{B}' \not\approx \mathcal{B}$. On the other hand consider $\mathcal{B}' = B_0 \cdot b_1 \cdot B_2 \oplus \{\bot\}$. In this case there is no such interference and the maximal promotion sequences of $\mathcal{B}'$ and $\mathcal{B}$ are the same; hence $\mathcal{B}' \approx \mathcal{B}$.

The next set of theorems explores what happens to promotion sequences as a fern is built up using FRONS and CONS. It turns out that if $\mathcal{A}$ = FRONS($\bot$,Nil) then EUREKA($\mathcal{A}$) = Ø and $\mathcal{A}$ has only empty promotion sequences; hence $\mathcal{A} <$ Nil. As we expect, EUREKA avoids $\bot$ and so the promotion sequences don't change with $\bot$ in the fern.

<u>Corollary 3.2</u>: If $\mathcal{A} <$ Nil then $\mathcal{A} \approx$ Nil.

In general, however, FRONS($\bot$,$\mathcal{A}$) $\not\approx \mathcal{A}$, but a weaker relationship can be established. We explore this fact in the general case of FRONS and later study CONS.

<u>Theorem 4</u>: If $\mathcal{A} < \mathcal{B}$ then FRONS(d,$\mathcal{A}$) $<$ FRONS(d,$\mathcal{B}$).

Now we can justify the use of the congruence symbol, $\approx$.

<u>Corollary 4.1</u>: If $\mathcal{A} \approx \mathcal{B}$ then FRONS(d,$\mathcal{A}$) $\approx$ FRONS(d,$\mathcal{B}$).

Therefore, $\approx$ acts like a left congruence. More conventionally we would choose an infix "binary" operator, $\otimes$ = FRONS, as the operator preserving this congruence.†

<u>Theorem 5</u>: FRONS($\bot$,$\mathcal{A}$) $< \mathcal{A}$.

We are tempted by the discussion preceding Corollary 3.2 to establish $\bot$ as an analogous "left identity". That would follow from a result symmetric to Theorem 5, which is not available. Unfortunately, promotion sequences of $\mathcal{A}$ can be disturbed by "FRONSing" a $\bot$: Consider

$\mathcal{A}$ = FRONS($\bot$,FRONS($\bot$,Nil)).

$\mathcal{B}$ = CONS($\bot$,Nil) has the promotion sequence $\langle \bot, \text{Nil} \rangle$ of length one, but EUREKA($\mathcal{A}$) = Ø. Thus $\mathcal{A} < \mathcal{B}$ but not $\mathcal{B} < \mathcal{A}$. We can, however, establish an idempotence result somewhat analogous to Theorem 5.

<u>Lemma 6.1</u>: Let $\mathcal{A} \in F$ then if $\forall i$ $a_{2i+1} \neq \bot$ then $\mathcal{A} <$ FRONS($\bot$,$\mathcal{A}$)

---

† An equivalence relation $\equiv$ is a <u>left</u> <u>congruence</u> with respect to an operator $\otimes$ if $x \equiv y$ implies $z \otimes x \equiv z \otimes y$.

Lemma 6.2: Let $\mathbf{A} \in F$ and k be the smallest integer such that $a_{2k+1} = \bot$. If $k = \omega$ or if $A_{2k}(\bot) > 1$ or if there exists $j < k$ such that $A_{2j}(\bot) > 0$, then $\mathbf{A} < \text{FRONS}(\bot,\mathbf{A})$.

We now have our idempotence results.

Theorem 6: $\text{FRONS}(\bot,\mathbf{A}) < \text{FRONS}(\bot,\text{FRONS}(\bot,\mathbf{A}))$.

Corollary 6.3: $\text{FRONS}(\bot,\mathbf{A}) \approx \text{FRONS}(\bot,\text{FRONS}(\bot,\mathbf{A}))$.

Corollary 6.4: $\lambda y.\text{FRONS}(\bot,y)$ is idempotent with respect to $\approx$.

Thus, FRONSing $\bot$ once is the same as FRONSing it repeatedly.

Theorem 7: If $d \neq \bot$ then $\text{FRONS}(d,\mathbf{A}) < \mathbf{A}$ iff $\exists k \leq \omega\ (\forall j < k\ a_{2j+1} = d\ \&(\Sigma A_{2i}(d) = \omega \text{ where } 0 \leq i \leq k)\ )$.

Corollary 7.1: Let $\mathbf{B} = \text{FRONS}(d,\mathbf{A})$. Then $\mathbf{B} < \mathbf{A}$ iff $\mathbf{B} \approx \mathbf{A}$ or $d = \bot$.

Now that we have thoroughly explained the behavior of FRONS under the interpretation above, we present similar results about the more familiar CONS operation. We, however, lose the idempotence.

Lemma 8.1: $\text{EUREKA}(\text{CONS}(d,\mathbf{A})) = \{\langle d,\mathbf{A}\rangle\}$.

Theorem 8: If $\text{CONS}(d,\mathbf{A}) < \mathbf{A}$ then $(\Sigma A_{2i}(d) = \omega \text{ where } 0 \leq i \leq \omega)$.

Corollary 8.2: Let $\mathbf{N} = \text{CONS}(d,\mathbf{A})$. Then $\mathbf{N} < \mathbf{A}$ iff $\mathbf{N} \approx \mathbf{A}$.

Nevertheless, $\approx$ is [illegible] a left congruence under CONS.

Theorem 9: If $\mathbf{A} \approx \mathbf{B}$ then $\text{CONS}(d,\mathbf{A}) \approx \text{CONS}(d,\mathbf{B})$.

Theorem 10: If $\langle d,\mathbf{B}\rangle \in \text{EUREKA}(\mathbf{A})$ then $\text{CONS}(d,\mathbf{B}) < \mathbf{A}$.

Semantics

In this section we present a semantics for a language that includes ferns as data structure for the user. Most of the axioms are conventional -- lifted from Stoy [19]. Ferns are embedded in an unconventional way, however, outside that tradition.

Two perspectives, operationally motivated, are necessary to understand why we refrain from using traditional power domain approaches to indeterminism. First is the call-by-need [21] or call-by-delayed-value [20] approach to parameter passing and to data structure construction [2]. We intend that no expression ever be evaluated until necessary to further the fabrication of an outermost result. In delaying evaluation we include specifically the content of ferns and of environments (treated also as objects -- albeit function-like). Such content, therefore, may be shared through borrowed references before it is ever evaluated. Upon first access such postponed evaluation proceeds at most once. There is no loss in such an operational protocol compared to that of conventional substitution for <u>deterministic</u> programs; we are considering more.

The second perspective, then, is the necessity to assure consistent handling of twice-used results of indeterminism. A major contribution of this paper is the two-step approach to this requirement. In the axioms which follow, indeterminism is introduced via Axiom 11 (FRONS) and is used via Axiom 12. In a later section we use these same axioms to reach a more comfortable solution to "twiceness", embedding all choice within ferns.

<u>Axioms</u>

$\mathcal{E}[\![I]\!]\rho = \rho[\![I]\!]$ where $I \in Id$	(1) Identifier
$\mathcal{E}[\![true]\!]\rho = tt$	(2) Truth
$\mathcal{E}[\![false]\!]\rho = ff$	(3) Falsehood
$\mathcal{E}[\![nil]\!]\rho = Nil$	(4) Empty fern
$\mathcal{E}[\![E_0 = E_1]\!]\rho = (\mathcal{E}[\![E_0]\!]\rho = \mathcal{E}[\![E_1]\!]\rho)$	(5) Equality
$\mathcal{E}[\![strictify(E_0,E_1)]\!]\rho = (\mathcal{E}[\![E_0]\!]\rho{=}\bot \rightarrow \bot, \mathcal{E}[\![E_1]\!]\rho)$	(6) Sequencer

The <u>strictify</u> operator is a sequencing operator, which is used to control the convergence properties of indeterminate structures. We also specify a conditional[†]

$\mathcal{E}[\![\underline{if}\ E_0\ \underline{then}\ E_1\ \underline{else}\ E_2]\!]\rho = \mathcal{E}[\![E_0]\!]\rho \rightarrow \mathcal{E}[\![E_1]\!]\rho, \mathcal{E}[\![E_2]\!]\rho$	(7) Conditional

Function invocation is usually conventional, although Axiom 12 is related in an unusual way.

$\mathcal{E}[\![\lambda I.E]\!]\rho = \lambda x.\mathcal{E}[\![E]\!]\rho[x/I]$	(8) λ-abstraction
$\mathcal{E}[\![E_0\ E_1]\!]\rho = \mathcal{E}[\![E_0]\!]\rho\ \mathcal{E}[\![E_1]\!]\rho$	(9) Application

---

[†] More cleanly we might have defined an additive conditional [19] and then defined "$\underline{if}\ E_0\ \underline{then}\ E_1\ \underline{else}\ E_2$" = "strictify($E_0$,adif E_0 E_1 E_2)" but we do not need additive conditionals in this paper [4].

$$\mathcal{E}[\![\text{cons}(E_0,E_1)]\!]\rho = \text{CONS}(\mathcal{E}[\![E_0]\!]\rho, \mathcal{E}[\![E_1]\!]\rho) \qquad \text{(10) Determinate Constructor}$$

We use <u>μfrons</u> locally in the user language to specify a fern which appears to <u>mu</u>tate under Axiom 12. In the next section we define another user primitve <u>frons</u> from <u>μfrons</u>.

$$\mathcal{E}[\![\mu\text{frons}(E_0,E_1)]\!]\rho = \text{FRONS}(\mathcal{E}[\![E_0]\!]\rho, \mathcal{E}[\![E_1]\!]\rho) \qquad \text{(11) Indeterminate Constructor}$$

While <u>cons</u> and <u>μfrons</u> are themselves always deterministic, they build ferns which can only be accessed using Axiom 12.

$$\mathcal{E}[\![\varepsilon IJ.E]\!]\rho = \lambda A.(\mathcal{E}[\![E]\!]\rho[d/I,\mathcal{B}/J] \text{ where } \langle d,\mathcal{B}\rangle \in \text{EUREKA}(A)) \qquad \text{(12) } \varepsilon\text{-abstraction}$$

Axiom 12 uses the call-by-need perspective on environment -- that arguments are only evaluated once -- to bind two variables <u>consistently</u>. The ε-abstraction borrowed from λ-abstraction and "ευρηκα" specifies that <u>one</u> choice of a pair is made on each application, and that these two items simultaneously and consistently become the values to which two variables are bound.

Delayed evaluation (call-by-need) may postpone this simultaneous binding, but it is not possible to use call-by-name which might separate it. For example, a call-by-name protocol might allow

$$\mathcal{E}[\![(\varepsilon IJ.\mu\text{frons}(I,J))\ F]\!]\rho$$

where $\mathcal{E}[\![F]\!]\rho = \mathcal{F} = \{tt,ff\}$
to yield $\{ff,ff\}$
if I were bound to ff and J were bound to $\{ff\}$ <u>ind</u>ependently. (But
$\text{EUREKA}(\mathcal{F}) = \{\langle tt,\{ff\}\rangle, \langle ff,\{tt\}\rangle\}$.)

Axiom 12 does not allow the substitution to distribute the choice across any other operator. Another way to perceive the situation is that the choice is made using call-by-need in the "environment" built on application of an ε-abstraction.

Thus, the conventional power domain constructions [16,18] do not work here. The EUREKA-image of a fern, not in general effectively computable, is only used as the domain of a choice function in Axiom 12. There, however, two bindings are effected in a necessarily consistent fashion, precluding call-by-name.

Examples

The examples in this section fall into three categories: simple, recursive on D, and recursive on $D + F$. The simple functions are amb [12] and arbit [8]. The recursive on D examples are the infinite sequence of integers, nn; a permutation of the integers; sequential or[13,pg 58] symmetric or[14,pg416]. The recursive on $D + F$ example merge streams.

The use of recursion in the programming language deserves some justification. Since we have not established monotonicity or continuty results we cannot use least-fixed-points to guarantee a meaning for recursion. Instances of λ-abstractions are in the language, however, so from a syntactic perspective we can construct Curry's Y-combinator and apply it to functions to admit recursive examples [19 , p.73] (solving no equations) under normal order evaluation, which is provided through call-by-need.

This scheme allows us to specify the construction of infinite ferns, but never to actually construct one. As long as A_0 is finite for any fern A, moreover, we can use a scheduling strategy to compute the elements of A_0 in parallel in order to find one member of EUREKA(A). If A_0 is infinite, then the search for an element of EUREKA(A), for $a \neq \bot$, is more complicated; a scheduling strategy akin to the conventional enumeration of the rational numbers suffices.

We first present fern probing relations:

first = εIJ.I;

rest = εIJ.J.

Since first(F) and rest(F) can have as many possible values as the cardinality of EUREKA(F) and these can be chosen independently on each application, it is difficult to use these probes on ferns built directly by μfrons. We, therefore, don't use first and rest here; in the next section this restriction will be relaxed.

The simple functions are non-recursive and include amb and arbit. Amb is a function which takes two arguments and returns a non-⊥ one of them if it exists.

amb = λA.λB.(εXY.X) μfrons(A,μfrons(B,nil)) .

Arbit is similar to amb, but it returns tt if the first one is non-⊥, false if the second one is non-⊥, and ⊥ otherwise.

arbit = λA.λB. amb strictify(A,true) strictify(B,false)

Of the functions that are recursive on D we have the natural numbers, nn, and the sequential and symmetric or.

nn = λI.C(I,nn(I+1))

If C is cons then we have an infinite sequence; if C is μfrons each of its promotion sequences is a permutation of the natural numbers.

```
or = λF. if null F then false
         else (εUV. if U then true else or V) F .
```

If a list is bound to F then the behavior is the sequential or; if a multiset is bound to F then the behavior is the symmetric or. Or has meaning for any fern, not just lists and multisets.

We next present one piece of code, which may be interpreted as four different programs. One of these is the "interrupt handler" or "input driver" required to service any active terminal in the airline reservation problem [1, 23 ,].

Before presenting these examples we extend D, hitherto an arbitrary domain, to include ferns.† This provides for ferns of ferns, sublists, and other nested data structures as anticipated by McCarthy [12].

Consider the code

```
merge = λM. if null M then nil
            else (εLR. if null L then merge R
                       else (εUV. C(U, merge K(V,R))) L) M.
```

where *C* and *K* are the constructors μfrons or cons.

† Showing that *F* is a domain is beyond the scope of this paper.

This code is quite similar in effect to code for flatten when *C*=cons=*K*. Such a function reduces a list of lists (or a matrix) into a single list of its second level elements (or a vector). When the argument is less well ordered more interesting things happen.

Let us assume that each sublist of L (or row of the matrix) has been constructed with Landin's prefix* [11], a version of cons which is strict in its first argument. Such sublists, or streams, are characteristic of serial input lines in a communication network. If these streams are composed into a multiset of substreams using μfrons, then we have the analog of a matrix with order and "seriality" with each row, but whose rows occur in an unspecified order.

The effect of *C*=cons=*K* is then to append all streams, the streams to be chosen in an order depending on the (temporal) order of convergence of the first elements in the streams. Once a stream is chosen to be first, however, it must be exhausted before another one is chosen to be merged in.

The effect of *C*=cons and *K*=μfrons is the desired interrupt handler. Upon each recursion the multiset structure is restored so that the output is a true merging of convergent prefixes of active streams. The order of the merge is determined.

If C=<u>μfrons</u>=K, the result is a multiset of all elements in all streams without regard to order either within streams or among streams. The stream nature of input, however, tends to assure that internal order within a stream will be reflected in the shuffled output, but this observation is based solely on operational behavior rather than required semantics.

If C=<u>μfrons</u> and K=<u>cons</u> then the shuffling of the previous paragraph is restricted up to the first infinite stream encountered. C assures that the result is an unordered structure, but K forces a selected stream to be exhausted before admitting the next one into the shuffle.

<u>Coping with anomalous behavior -- frons</u>

The programmer working with indeterminism and the language of the previous section has sufficient power to solve some interesting problems, but he has some irritation as well. For example, let $\mathcal{E}[\![E]\!]\rho = \mathcal{F}$ be a fern and let EUREKA($\mathcal{F}$) have two elements $\langle d,\mathfrak{G}\rangle$ and $\langle d',\mathfrak{G}'\rangle$. Now within the scope of one application of ε, say

$$\mathcal{E}[\![\varepsilon IJ.(I = I)\ E]\!]\rho$$

the values of I and J are consistently paired, so that this evaluation yields <u>tt</u> always: either I is bound to d or to d' exclusively. However, every application of ε allows a new choice for the pair:

$$\mathcal{E}[\![(\varepsilon IJ.I\ E) = (\varepsilon IJ.I\ E)]\!]\rho$$

may evaluate either to <u>ff</u> or to <u>tt</u> depending on whether d and d' are chosen by either application of ε or not. That the equality of this example might not hold should be quite disturbing to the applicative programmer because the syntax is identical on either side of the equal sign.

The headaches compound if one grapples with "fairness." A fair implementation of Axiom 12 would have to select an element from a fern's EUREKA image with probability equal to all the rest. Consider the infinite multiset of natural numbers; a possible result of $\mathcal{E}[\![\lambda F.F\ nn(0)]\!]\rho$:

$$\mathcal{E}[\![F]\!]\rho = \mathbb{N} = \lambda x.i \in F(\omega).$$

Then every fair evaluation

$$\mathcal{E}[\![(\varepsilon IJ.I)\ F]\!]\rho$$

should turn up "yet another" integer; this should act like a random integer generator* which is not effectively computable. Fairness is, therefore, hopeless.

---

* This observation about fairness and multisets is due to Robert E. Filman.

Recalling our early discussion of "twiceness" and call-by-need, our programmer perceives that the problem lies in repeated applications of Axiom 12 to the same fern, yielding (legally) inconsistent results and precluding fairness. Theorem 10 shows a way out: fix it so that the application of ε is embedded <u>within</u> a fern. Then the restrictions on evaluating arguments once will apply Axiom 12 only once per fern:

$$frons = \lambda X.\lambda Y.(\epsilon IJ.cons(I,J))\ \mu frons(X,Y).$$

By Theorem 10 $\mathcal{E}[\![frons(X)Y]\!]\rho < \mathcal{E}[\![\mu frons(X,Y)]\!]\rho$ so that <u>frons</u> is a correct, but weak replacement for <u>μfrons</u>; its behavior is <u>one</u> of the possible behaviors of <u>μfrons</u>.

In effect, a fern built with <u>frons</u> in place of <u>μfrons</u> everywhere, is built as a list incorporating <u>one</u> of the EUREKA choices possible at each step of a promotion sequence. That list reflects one, not all, maximal promotion sequences. Of course, the call-by-need convention serves to delay selection of precisely which one; the choice unfolds simultaneously with the need for such a sequence.

The anomalous behavior of repeated applications of ε, seen to yield inequality between syntactic identicals before, now disappears. Every application of ε yields consistent unique results, as if the consistency rules for ε have moved outside its scope. (Indeed, this is just what the <u>cons</u> in the definition of <u>frons</u> does: it freezes the ε-scope in which it is evaluated.) <u>Frons</u> is no more a function than is <u>μfrons</u>, but <u>first</u> and <u>rest</u> become deterministic functions in a style where <u>frons</u> displaces all uses of <u>μfrons</u>.

Moreover, the fairness of <u>frons</u> is an issue different from the fairness of ε. If <u>frons</u> replaces <u>μfrons</u> in all code (e.g. the examples of the previous section), then there is no longer any fairness debate on ε. All remaining occurrences of ε (aside from that in the definition of <u>frons</u> itself) are deterministic, so we need only concern ourselves with the fairness of <u>frons</u> (or that one occurrence of ε). An important refinement has been made if ε is only indeterministic within <u>frons</u>: we know that ε is never necessarily applied to the same fern, A, twice unless $A_0 = \{a_1\}$ a singleton.

Thus, a concern for fairness need <u>not</u> account for fair relational (as opposed to functional) behavior of Axiom 12 choosing from the EUREKA image of one fern repeatedly. Under <u>frons</u> only one indeterminate choice is ever made for every fern, so that fairness need never be defined over independent applications of Axiom 12; instead we can establish some dependency according to the order of the maximal promotion sequence selected. Just such a proposal was the original goal of our work [5 , appendix].

Conclusion

We have developed a theoretical perspective on indeterminism for applicative multiprogramming. There have been several other approaches to incorporating indeterminism in pure applicative languages. Of these, the earliest is McCarthy's amb operator [12] which returned one or the other of its two arguments, avoiding ⊥ as a value when possible. Bottom-avoidance is also the essence behind Kosinski's arbiter [10] and Keller's arbit [8]. None of these extends to consideration of fairness.

By embedding the indeterminism within a structure -- the fern -- we allow references to the contention to be passed as arguments, establishing shared references to the choice before it is made. Call-by-need protocol for application and for structure definition not only allows reference to an unmade (but specified) choice, but also guarantees normal order evaluation and allows us recursive definitions using the Y combinator,

Encapsulating indeterminism inside an object allows us to attack fairness of that object, rather than having to relate fairness to overall control structure. If choice is made fairly as an object is built up and torn down -- over a period of time -- then the system has a fair choice mechanism available to it. In the preceding section we only anticipated solutions to fairness; here we offer a conjectural definition of what a fair implementation of ferns should do.

Fairness Principle : An interpretation of cons and μfrons may be said to be fair if there exists $f: F \to \omega$, some monotonically increasing function with respect to $<$ on F and to the total order on ω, such that $f(A)$ is bounded by the length of all maximal promotion sequences of A and f meets the following requirement. When $\mathcal{E}[\![F]\!]\rho = A_0$ with maximal promotion sequence $\{\langle d_i, A_i\rangle\}_{i>0}$ and $\mathcal{E}[\![\mu\text{frons}(d,F)]\!]\rho = B_0$ with maximal promotion sequence $\{\langle e_i, B_i\rangle\}_{i>0}$ then either $d = \bot$ and $\forall i(e_i = d_i)$ or $d \neq \bot$ and $e_i = d_i$ for $i < f(A_0)$, $e_{f(A_0)} = d$, and $e_{i+1} = d_i$ for $i > f(A_0)$.

The fairness principle conjectures a function which identifies the prefix of a promotion sequence which is not altered by μfronsing new elements into the fern. Thus, new values are inserted after such an (ever increasing) prefix. This guarantees that every element in a fern will occur "early" in a maximal promotion sequence regardless of how many items are adjoined to that fern with μfrons; eventually the new values will take their place after already extent elements. Eventually any non-⊥ element occurs "first" as the fern is traversed using Axiom 12. (The conjectured function, f, need not be locally properly increasing, but it must be asymptotically increasing for fairness to hold.) We have implemented one such scheme [5] using timestamps (birthdates) to effect the function f.

In summary, we have defined ferns as a data structure encapsulating indeterminism by extending lists. We have proven several results regarding the mathematical properties of ferns, particularly relating to the idempotency resulting from adding a contending process that doesn't behave; semantics of list structures are retained, however. Following a semantics for a language incorporating ferns we presented a few examples; others are available elsewhere [6,5]. Having stated the case for ferns carrying indeterminism within applicative languages, we offered a simple refinement to solve the twiceness anomaly of other formulations of indeterminism in applicative languages. That twist allows us to confront issues of fairness, set forth just above.

The challenge of fairness is that contention of processes in an applicative system could neither be ignored nor honored without any explicit scheduler to assure such effect. It is this challenge which we expect ferns to meet. Future work will refine the concept of fairness and reconcile it within a generalized formulation of denotational semantics.

Acknowledgement: We thank Steven D. Johnson and particularly Mitchell Wand for numerous constructive discussion 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 and David MacQueen who offered some thoughtful proposals on earlier versions of this paper.

REFERENCES

1. 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.

2. Friedman, D. P., and Wise, D. S. CONS should not evaluate its arguments. In S. Michaelson and R. Milner (eds.), Automata, Languages and Programming, Edinburgh Univ. Press, Edinburgh (1976), 257-284.

3. Friedman, D. P., and Wise, D. S. Aspects of applicative programming for parallel processing. IEEE Trans. Comput. C-27, 1 (April, 1978), 289-296.

4. Friedman, D. P., and Wise, D. S. A note on conditional expressions. Comm. ACM 21, 11 (November, 1978), 931-933.

5. Friedman, D. P., and Wise, D. S. An approach to fair applicative multiprogramming. In G. Kahn and R. Milner (eds.), Proc. of Intl. Symp. on Semantics of Concurrent Computation, Springer, Berlin (1979), to appear.

6. Friedman, D. P., and Wise, D. S. An indeterminate constructor for applicative programming. Proc. 7th ACM Symp. on Principles of Programming Languages (1980), 245-250.

7. Henderson, P., and Morris, J. H., Jr. A lazy evaluator. Proc. 3rd. ACM Symp. on Principles of Programming Languages (1976), 95-103.

8. Keller, R. M., Lindstrom, G., and Patil, S. Data-flow concepts for hardware design. COMPCON Conf. (1980),

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

10. 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.

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

12. 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.

13. 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.

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

15. MacLane, S. Categories for the Working Mathematician, Springer-Verlag, New York (1971) 26.

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

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

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

19. Stoy, J. E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, M.I.T. Press, Cambridge, MA (1977).

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

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

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

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

Laboratory for Programming Methodology

Symposium on Functional Languages and Computer Architecture

Invited Papers