Z
699
C3
no. 205

UNIVERSITY OF CALIFORNIA

Irvine

An Applicative Computing Language

TR# 205

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Information and Computer Science

by

Joseph Paul Minne

Committee in charge:

Professor Lubomir Bic, Chair

Professor Fred M. Tonge

Professor George S. Lueker

Professor Kim P. Gostelow

1983

The dissertation of Joseph Paul Minne is approved,

and is acceptable in quality and form for

publication on microfilm:

[illegible]

Fred M. Tonge

[illegible]

[illegible]

Committee Chair

University of California, Irvine

1983

DEDICATION

This dissertation is dedicated to my family,

Wendi Anne Georges Minne,

Paul and Judith Minne,

and

Joseph and Magdalena Nagy.

Their love and support were invaluable.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I am indebted to my advisor, Kim P. Gostelow. His patient guidance and helpful insight were valuable contributions to this work. I also wish to thank Lubomir Bic, Fred Tonge, and George Lueker for reviewing and commenting upon drafts of this dissertation. Finally, I wish to express my appreciation to Arvind, Wil Plouffe, Bob Thomas, and Jim Neighbors, for their criticism and helpful suggestions.

VITA

1973	B.S. in Mathematics and Physics, University of Southern California, Los Angeles
1973-1975	Teaching Assistant, University of California, Irvine
1975-1976	Senior Mathematical Analyst, Manufacturing and Consulting Services, Inc., Costa Mesa, California
1976-1978	Research Assistant, University of California, Irvine Area: Dataflow Architecture
1978-1979	Instructor at University of California, Irvine
1980-1982	Assistant Professor of Mathematics and Computer Science at The College of William and Mary, Williamsburg, Virginia
1982-1983	Member of the Technical Staff, Hughes Aircraft Company, Fullerton, California
1983	Ph.D. in Information and Computer Science, University of California, Irvine Dissertation: "An applicative computing language"

ABSTRACT OF THE DISSERTATION

An Applicative Computing Language

by

Joseph Paul Minne

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1983

Professor Lubomir Bic, Chair

Closed applicative languages (CALs) are a highly parallel and semantically appealing models of computation, but they do not address process and resource related issues in computing; i.e.:

- they do not express histories of computation.

- they cannot describe interprocess communication.

This dissertation defines a new model, CFFP, derived from a CAL, FFP, which addresses these issues. In CFFP, the semantics of FFP are relaxed to allow a computation to persist over a series of actions, and explicit cycles are added to allow bidirectional communication between loci of computation. CFFP retains the appealing characteristics of CALs, and addresses process and resource related computing issues.

CHAPTER 1

Introduction

1.1 Weaknesses of von Neumann computation

John von Neumann [10] developed the model which characterizes conventional computation over thirty years ago. This model divides a computation into a series of primitive instructions, and sequentially executes these instructions in a central processing unit (CPU). A separate memory unit holds the instructions and their data until execution, and accepts the results afterward. Information passes between these two units one element (instruction or datum) at a time[1].

Several persons have expressed dissatisfaction with the von Neumann model, and advocate alternatives to it.[2] Their objections arise from at least two factors:

1. The cost of fabricating processing elements[3] has

[1]This description essentially characterizes conventional computation even though it may diverge from the specific description of a particular computing machine.

[2]These persons include Arvind, Gostelow, and Plouffe [1, 3], Backus [6], Berkling [9], Dennis [16], Frank [17], Friedman and Wise [21], Keller, Lindstrom, and Patil [25], Landin [26], Mago [27, 28], Tonge and Cowan [34], and Turner [35].

[3]Processing elements are devices capable of executing a primitive instruction.

decreased substantially as time has passed. This reduced cost urges exploitation of the parallelism found in most problems, and decries the sequentiality of the von Neumann model.

2. The semantics of von Neumann systems are unwieldy. Functions and compositions of functions acting on values directly represent computations in the CPU, but shipping the values to and from memory[4] requires extra domains and new functions (see Stoy [33]). These new elements evoke hierarchies of functions and complex computation rules.

These factors argue strongly for an alternative model.

The class of closed applicative languages (CALs) [5] is such a model. In CALs, programs are expressions composed of independent sub-expressions, computations (the loci of activity in the program) are distinguished expressions (applications), and a separate memory is not an integral part of the system. The independence of sub-expressions promotes parallelism, and the absence of non-computational elements eliminates the need for complex semantics.

---

[4]In higher-level programming languages, assignment statements and variables often represent the shipping mechanism.

1.2 Computing systems and applicative computation

A computer must accomplish many tasks in addition to computation. At the very least, we expect it to communicate with us, and to manage a collection of associated devices (e.g., card readers, line printers, and tape drives). In general, it must coordinate the collection of concurrent processses and logical resources typical of a modern time-sharing environment. Any successful model of computing must model these activities in addition to modelling computation. In the remainder of this dissertation, computation refers to the functional combination of values, and computing refers to the full range of computations and resource-sensitive activities.

A computing system might incorporate CALs (or close CAL-variants) in several different ways. An ideal incorporation would represent the entire system and each process as a computation, and would embed each resource within a supervising computation. This direct incorporation system makes no distinction between computation and other aspects of computing; thus, it preserves all desirable properties of CALs.

The direct incorporation method cannot directly employ the notion of computation expressed in CALs. Systems, processes, and resource managers (specialized forms of processes) constitute a set of asynchronously interacting entities executing a computation over time. In a closed applicative language, computations interact in a strictly hierarchical manner, and have no temporal persistence. These restrictions are too severe for unmodified CALs to serve as the computational basis in a direct incorporation model of computing systems.

Other methods of incoporating CALs in a computing system also seem fraught with difficulty. All strictly computational methods must reconcile the restrictions of CALs with the requirements of computing. Separating computation from other aspects of computing results in asymmetric systems which cannot address the interaction of two or more ongoing processes without obscuring or ignoring the nature of those processes. Careful alteration of the basic computational model would seem to hold the greatest promise for producing a satisfactory model of computing.

This dissertation produces a model of computing based on CALs.

- The remainder of this chapter gives a detailed description of CALs.

- Chapter 2 describes the origins of CALs and surveys research related to that of this dissertation.

- Chapters 3 and 4 modify a CAL to produce a computing language.

- Chapter 5 concludes this work with some suggestions for future study.

1.3 Closed applicative languages

Closed applicative languages (CALs) have four major components: constructor syntax, distinguished constructors, meaning functions, and representations. The basic meaning function is identical for each CAL, but the other three components may vary within the constraints described below.

- Constructor syntax.

A constructor syntax for a set of expressions, $\underline{E}$, consists of a set of atoms, $\underline{A}$, and a set of constructors, $\underline{K}$, which together define $\underline{E}$. The atoms are the primitive expressions of $\underline{E}$, and the constructors map sequences of expressions into new expressions. Each constructor operates on sequences of a specific length, $n \geq 0$, and is a function from some subset of $\underline{E}^n$ into $\underline{E}$.

Any expression, e, is either an atom or the result of some constructor, k, acting on some sequence, [el,...,en]. If e is an atom, then it is not a construct. If e is a construct, then its construction is unique.

- The distinguished constructor.

  A CAL distinguishes one constuctor in $\underline{K}$ for special treatment by the meaning function. This constructor must be a total function on $\underline{E}^2$. "ap" commonly denotes this constructor, and an "application" is the result of ap acting on a pair of expressions. Applications are the seat of computation in CALs.

- The meaning function.

  The value returned by the meaning (or semantic) function, $\underline{m}$, is the meaning of any meaningful expression. ($\underline{m}$ is undefined for meaningless expressions.) These meanings are also expressions, and they evaluate to themselves. The set of all meanings is also known as the set of constants, $\underline{C}$.

  The meaning of an atom is itself. The first step in evaluating a constructed expression is the evaluation of its subexpressions. If the construction is not an application, its meaning is the original constructor applied to the sequence of expressions resulting from the first step. If the construction is an application, its further evaluation requires the use of the representation function.

- The representation function.

Each constant expression represents a function which maps constants into expressions. The representation function, $\underline{r}$, returns that function, i.e.,

$$\underline{r}:\underline{C}\text{-->}\{\underline{C}\text{-->}\underline{E}\} \quad .$$

Given constant expressions f and x, the meaning of their application, ap[f,x], is the meaning of the expression resulting when the function represented by f acts on x. If f or x are not constants, the meaning function must evaluate them as required for any construction, before $\underline{r}$ or its result is applicable[5]

$$\underline{m}\text{ap}[f,x] = \underline{m}\{\{\underline{r}\{\underline{m}f\}\}\{\underline{m}x\}\}$$

In addition to their basic elements, CALs also posses a transition function, $\underline{t}$, which maps $\underline{E}$ into $\underline{E}$. In general, $\underline{t}$ implements $\underline{m}$ by defining a set of executable

[5]Throughout the following discussion, the result of applying a function to an expression will be denoted by juxtaposition. Set-brackets "{ }" will often associate a function with its argument expression, but this notation has no syntactic or semantic significance beyond clarifying the operational association.

subexpressions and allowing an arbitrary one of them (if any exist) to act. If no such subexpression exists, the result is the original expression. The executable set defines the degree of parallelism in an expression, and repeated applications of $\underline{t}$ convert meaningful expressions into their meanings.

For CALs, $\underline{t}$ takes a constant expression to that same constant, and takes non-constant expressions to copies of themselves with some one innermost application replaced by its execution. An innermost application ap[f,x] contains no sub-applications, and its execution is the action of the function represented by the first subexpression of the application $\underline{r}$f on the second x. (We often call the first subexpression the function-part, and the second the argument-part.) A given transition may select any innermost application for replacement.

The transition function complements the meaning function. The semantic function defines the meaning of a meaningful expression, and the transition function, acting on all expressions, defines an alternate mechanism for finding that meaning. For any meaningful expression e, repeated application of $\underline{t}$ produces a unique constant, c, which is the meaning of e. More formally,

$$\forall e \in \underline{E}: \ \underline{m}e=c \iff \exists n \geq 0 : \underline{t}^n e = c \in \underline{C} \quad .$$

e-->e' denotes the transition of e into e', and e-->>e''

$CAL == (\underline{E}=(\underline{A},\underline{K}), ap\in\underline{K}, \underline{C}, \underline{m}, \underline{r})$

1. $\underline{A}\subseteq\underline{C}\subseteq\underline{E}$
2. $k\in\underline{K} \Rightarrow \exists n\geq 0, E'\subseteq\underline{E}^{n}: k:E' \dashrightarrow \underline{E}$
   - 2.1 $k\neq ap,\ ci\in\underline{C},\ e=k[c1,\ldots,cn]\in\underline{E} \Rightarrow e\in\underline{C}$
   - 2.2 $c\in\underline{C}, c\notin\underline{A} \Rightarrow \exists k\neq ap, ci\in\underline{C}: c=k[c1,\ldots,cn]$
3. $e\in\underline{A} \Rightarrow \forall[k:E'\dashrightarrow\underline{E}]\in\underline{K}, [e1,\ldots,en]\in E': e\neq k[e1,\ldots,en]$
4. $e\in\underline{E}$ & $e\notin\underline{A}$
   - $\Rightarrow$ 4.1 $\exists k\in\underline{K}, [e1,\ldots,en]: e=k[e1,\ldots,en]$
   - 4.2 $h\neq k \Rightarrow \forall[x1,\ldots,xm]: e\neq h[x1,\ldots,xm]$
   - 4.3 $[x1,\ldots,xn]\neq[e1,\ldots en] \Rightarrow e\neq k[x1,\ldots,xn]$
5. $\exists\underline{M}\subseteq\underline{E}: \underline{m}:\underline{M}\dashrightarrow\underline{C}$ & $\underline{C}\subseteq\underline{M}$
6. $\exists\underline{t}:\underline{E}\dashrightarrow\underline{E} \ni \{\forall e\in\underline{E}: \underline{m}e=c \Longleftrightarrow \exists n\geq 0: \underline{t}^{n}e=c\in\underline{C}\}$
7. $\forall k\in\underline{K}, k\neq ap: \underline{m}k[e1,\ldots,en] = k[\underline{m}e1,\ldots,\underline{m}en]$
8. $\underline{r}:\underline{C}\dashrightarrow\{\underline{C}\dashrightarrow\underline{E}\}$
9. $\underline{m}ap[f,x] = \underline{m}\{\{\underline{r}\{\underline{m}f\}\}\{\underline{m}x\}\}$

Figure 1. CAL summary

denotes a sequence of transitions leading from e to e''.

Figure 1 summarizes our description of CALs. Axioms one through four express syntactic constraints, and five through nine semantic ones. These rules allow a choice of $\underline{A}$, $\underline{K}$, and $\underline{r}$ to specify a particular CAL.

1.4 Formal functional programming (FFP)

Formal functional programming is the best known closed applicative language. It embodies the principles of CALs in a form which we can use as the basis of a new language (developed in chapters 3 and 4) encompassing all the elements of computing. This new language will not be a CAL, but it will preserve the essential qualities of one.

1.4.1 The elements of FFP

A set of atoms, a set of constructors, and a representation function specify FFP among the CALs.

1. The set of atoms consists of any character-string. Some characters (e.g., "(", ")", blank, and ":") will not be used since such use would conflict with the notation for constructed expressions. The definition of K̲ will implicitly define these forbidden characters.

2. The set of constructors consists of "ap" and "bot" in union with an infinite set of sequence constructors $\{sn{:}n \geq 1\}$. bot is a special zero-place constructor which builds the formally undefined expression. Each sn takes a sequence of n expressions, not including bot[], to a formal FFP sequence of length n. Figure 2 defines our notation for these constructs.

   $f,x,ei \in \underline{E}$

   ```
   ap[f,x] = (f:x)
   bot[] = ⊥
   sn[el,...,en] = <el,...,en>   ei≠⊥
   ```

   Figure 2. Basic syntax for FFP

3. The representation function maps any constant to a function on constants. The effect of this

function on constants is as follows:

a. For $\perp$, $\underline{r}$ returns "U", the everywhere undefined function, which returns $\perp$ for any argument expression.

b. For any constant sequence, $\underline{r}$ returns a function which forms a new application. The function-part of this new application is the first element of the original sequence, and the argument part is the pairing of the original sequence and the original argument.

$\{\underline{r}\langle f1,\ldots,fn\rangle\}x == (f1:\langle\langle f1,\ldots,fn\rangle,x\rangle) \quad fi,x\in\underline{C}$

c. All atoms represent either primitive functions (primitive atoms) or programmer-defined functions (defined atoms). Initially, all atoms are primitive, and most of them represent U. Appendix I lists a set of primitive atoms which represent other functions together with those functions.

In an FFP environment, the user may execute the meta-expression

$$\text{DEF } a == c \quad a\in\underline{A}, c\in\underline{C}$$

to provide a new representation for a. This meta-expression indicates that atom a now represents the same function as expression c. This definition facility enhances the clarity of FFP computations, but does not alter their power.

For a primitive atom, r simply returns the appropriate primitive function. For a defined atom f, with definition g, rf is the same function as rg.

FFP employs one additional meta-function, d. For any primitive atom, d returns "#", and for any defined atom, d returns the appropriate defining construct.

Figure 3 expresses the semantic function of FFP in a form by Backus [6] based on McCarthy's conditional expressions [30].

```
e,ei,x,y,f,fi,g∊E

me == e∊A => e;
      e=<el,...,en> => <mel,...,men>;
      e=(f:x) => {f∊A => {df=# => m{{rf}{mx}};
                          df=g => m(g:x)};
                  f=<fl,...fk> => m(fl:<f,x>);
                  m(mf:x)};
      ⊥
```

Figure 3. Complete FFP semantics

This m invokes r only for primitive atoms. To distinguish primitive from defined atoms, and to retrieve the defining constructs of the latter, m invokes d. Figure 3 forms the semantic basis for the variations on FFP described in chapters 3 and 4.

1.4.2 Programming in FFP

Applications correspond to conventional programs. Their function-parts correspond to program bodies, and their argument-parts to program data. In the simplest case, function-parts are primitive atoms representing some basic operation (e.g., ADD), and the applications perform a straightforward calculation (e.g., (ADD:<2,3>) --> 5). More complex cases display a rich set of constructs for forming programs.

Sequences can act as program structures. (In FFP, one often refers to program structures as functional forms.) For example, <WHILE,p,f> acts as a while-loop, and <IF,p,f,g> serves as an if-then-else construction[6]; p stands for a predicate, and f and g represent general functions. In the while-loop, f serves as the loop-body, and in the if-then-else, f is the then-clause, and g is the else-clause. Another functional form is <CONS,f1,...,fn> which indicates the parallel execution of the fi. Initial

---

[6] p, f, and g are arbitrary constant expressions. Lower case letters denote unspecified expressions throughout this dissertation.

primitive atoms[7] determine a complete set of primitive FFP program structures. Appendix I includes the set of primitive atoms suggested by Backus [5, 6].

The use of certain notational conventions simplifies FFP programming. Figure 4 provides an alternative notation for five functional forms.

f,fi$\in$$\underline{E}$

[f1,...,fn]	<CONS,f1,...,fn>
f1*...*fn	<COMP,f1,...,fn>
'f	<CONST,f>
/f	<INSERT,f>
@f	<AA,f>

Figure 4. Notation for selected functional forms

In addition, three atoms have special significance. Functions which deal with sequences use "ϕ" as the sequence of zero length. Predicates and boolean functions regard "T" and "F" as the traditional boolean values.

ARBADD is a defined atom which demonstrates FFP programming. ARBADD represents a function adding either two numbers or two matrices of arbitrary congruent dimension.

DEF ARBADD == <IF,ATOM*1,ADD,@ARBADD*TRANS>

Evaluation of (ARBADD:<4,5>) exemplifies the operation of $\underline{r}$ARBADD on simple numbers.

$\mathbb{m}$(ARBADD:<4,5>)

[7]Henceforth, the term primitive atom refers to a primitive atom with function other than U.

produces

```
m(<IF,ATOM*1,ADD,@ARBADD*TRANS>:<4,5>)
```

by replacing ARBADD with its definition, and the FFP rule for the representation of a sequence in turn produces

```
m(IF:<<IF,<COMP,ATOM,1>,ADD,<COMP,<AA,ARBADD>,TRANS>>,
      <4,5>>) .
```

IF is a primitive function, and the rule for primitive functions results in

```
m{{rIF}<<IF,<COMP,ATOM,1,ADD,<COMP,<AA,ARBADD>,TRANS>>,
         <4,5>>}
```

where the evaluation of constant parts has been omitted. Executing the representation of IF creates

```
m((ATOM*1:<4,5>):<<ADD,<4,5>>,<<COMP,<AA,ARBADD>,TRANS>,
                                         <4,5>>>)
    = m(m(ATOM*1:<4,5>):<<ADD,<4,5>>,
                        <<COMP,<AA,ARBADD>,TRANS>,<4,5>>>)
```

Evaluating (ATOM*1:<4,5>) produces

```
m(COMP:<<COMP,ATOM,1>,<4,5>>) ,
```

and this results in

```
m(ATOM:(1:<4,5>)) .
```

(As in the last derivations, we will henceforth omit primitive executions of r in the interest of clarity and brevity.) The evaluation of (ATOM*1:<4,5>) concludes with

```
m(ATOM:m(1:<4,5>)) = m(ATOM:4) = T ,
```

and the evaluation of (ARBADD:<4,5>) becomes

```
m(ARBADD:<4,5>)
```

```
= 𝔪(T:<<ADD,<4,5>>,<<COMP,<AA,ARBADD>,TRANS>,<4,5>>>)
= 𝔪(ADD:<4,5>)
= 9  .
```

On two congruent matrices,

$$<<3,4,5>,<6,7,8>> = \begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

and

$$<<30,40,50>,<60,70,80>> = \begin{bmatrix} 30 & 40 & 50 \\ 60 & 70 & 80 \end{bmatrix} \quad ,$$

𝔪(ARBADD:<<<3,4,5>,<6,7,8>>,<<30,40,50>,<60,70,80>>>) acts in essentially the same manner as 𝔪(ARBADD:<4,5>), until the evaluation of 𝔪(ATOM*1:<<<3,4,5>,...>>) produces "F". This means that the total evaluation becomes:

```
𝔪(F:<<ADD,<<<3,4,5>,...>>>,
      <<COMP,<AA,ARBADD>,TRANS>,<<<3,4,5>,...>>>>)
 = 𝔪(@ARBADD*TRANS:<<<3,4,5>,...>>)
 = 𝔪(@ARBADD:𝔪(TRANS:<<<3,4,5>,...>>))
 = 𝔪(@ARBADD:<<<3,4,5>,<30,40,50>>,<<6,7,8>,<60,70,80>>>)
 = 𝔪(AA:<<AA,ARBADD>,<<<3,4,5>,<30,40,50>>,...>>)
 = 𝔪<(ARBADD:<<3,4,5>,<30,40,50>>),
     (ARBADD:<<6,7,8>,<60,70,80>>)>
 = <𝔪(ARBADD:<<3,4,5>,<30,40,50>>),
    𝔪(ARBADD:<<6,7,8>,<60,70,80>>)>   .
```

The two sub-evaluations may proceed in parallel to produce

```
<<𝔪(ARBADD:<3,30>),𝔪(ARBADD:<4,40>),𝔪(ARBADD:<5,50>)>,
 <𝔪(ARBADD:<6,60>),𝔪(ARBADD:<7,70>),𝔪(ARBADD:<8,80>)>>  ,
```

and the execution of these six evaluations results in

$$<<33,44,55>,<66,77,88>> = \begin{bmatrix} 33 & 44 & 55 \\ 66 & 77 & 88 \end{bmatrix} \quad .$$

The final six instances of m(ARBADD:...) indicate a high degree of explicit parallelism; they are independent evaluations. This parallelism reflects the independent operations performed in adding matrices. Conventional programming languages obscure and constrain such concurrency while FFP naturally expresses it.

1.5 Computational properties of CALs and other languages

Backus [5] asserts that closed applicative languages possess six important formal properties in addition to their natural expression of parallelism.

1. _Idempotency of meaning_. The meaning of an expression is also an expression, and such meanings always evaluate to themselves.

2. _The "anti-quote" property_. Constant expressions represent themselves, and representing values requires no special quoting mechanism.

3. _Non-extensionality_. The meaning of an expression is not the function it represents. As such, functional equality of expressions e and x does not guarantee fe = fx for an arbitrary function f.

4. _Single-type functions_. All allowable functions are mappings from the same domain onto the same

range. For CALs, these sets are constants and expressions respectively.

5. _The extended Church-Rosser property_.
   a. The basic property states that all terminating sequences of reductions on a given expression yield the same meaning. In CALs, an execution of the transition function corresponds to a reduction.

   b. The extension requires that all reduction sequences on a meaningful expression terminate.

6. _The reduction property_. All subexpressions of a meaningful expression are themselves meaningful, and if a given sub-expression is replaced by its meaning, the meaning of the total expression remains unchanged.

No other class of languages possesses all six of Backus' formal properties. Most programming languages do not possess _any_ of them; although expression-oriented languages have limited forms of the Church-Rosser and reduction properties.

LISP is the most familiar expression-oriented language. While it bears a superficial similarity to FFP, it does not distinguish its applications from its sequences. This ambiguity causes LISP to violate idempotency. In addition, LISP expressions employ variables, thereby violating the anti-quote property since variables stand for other values and not for themselves.

The lambda-calculus and combinators are the progenitors of both LISP and CALs. These older formal languages analyze the concept of a function, and their definitions of meaning rely on functional equivalence; i.e., they are extensional. In addition, the concept of reduction in the lambda-calculus is more complex than that found in CALs, which makes establishing and using the Church-Rosser and reduction properties more difficult. Combinators also suffer from this added complexity because they have multiple function-types.

CALs rely on three critical features in satisfying the six formal properties.

1. Applications are clearly distinct from all other constructs. Since applications correspond to computations, this feature guarantees idempotency.

2. There are no variables. This negates any need for a notion of quoting.

3. All expressions are based on a constructor

syntax, and meanings reflect this basis.

By retaining these three features, the computing language produced in chapters 3 and 4 preserves the fundamental character of the six formal properties.

CHAPTER 2

Approaches to applicative computing

2.1 Origins

Closed applicative languages derive from two systems of mathematical logic: the combinatory logic of Schonfinkel [31] and Curry [13, 14], and the lamba-calculus of Church [11]. The primitives of combinatory logic are an arbitrary set of free variables and a specified set of combinators; the free variables represent themselves, and the combinators stand for functions which act on any objects in the logical system. Combinatory logic forms new objects (combinations) by pairing (e.g., (A B) is a combination if A and B are combinations); by convention, A B C denotes the same combination as ((A B) C), and parentheses are often omitted when this precedence rule determines the intended expression. J is a typical combinator, and represents a function such that

J A B C D --> A B (A D C) .

Systems of combinatory logic are not closed applicative languages in the Backus sense since pairs may denote either constants or applications; if u, v, x, and y are free variables, (J x) denotes a combination which happens to be the constant (J x) while ((((J u)v)x)y) denotes an application with operator (((J u)v)x) and operand y; i.e., J u v x y executes to form u v (u y x).

The lambda-calculus replaces combinators with new constructs called lambda-expressions, formed by pairing a variable with an expression; if x and A are a variable and an expression, $\lambda x.A$ is a lambda-expression with bound variable x and body A. $\lambda x.A$ represents a function such that $(\lambda x.A)B$ returns A with all free occurences of x in A replaced by B. We can say an occurence of v in C is free if it is not part of a lambda-expresion contained in C with v as bound variable. $\lambda v.B$ is _well-formed_ only if B is well-formed and contains at least one free occurrence of v, and B is well-formed if it is a variable, a pair of well-formed expressions, or a well-formed lambda expressions.

We may now express J as

$$J = \lambda a.\lambda b.\lambda c.\lambda d.ab(adc) \ .$$

The lambda-denotation of functions avoids the ambiguity found in the pairs of combinatory logic; now, any pair with a lambda-expression in the operator position is an application, and any expression with no such pair denotes a constant. The denotation of functions by lambda-expressions eliminates the lambda-calculus as a CAL since several lambda-expressions may denote the same function,

$$\lambda x.x \quad \lambda y.y \quad \lambda a.a \quad \lambda b.b \quad \lambda z.z$$

and exactly one of them is the standard form. Thus, the others are not constants in the CAL sense, and their evaluation violates the CAL rule that the meaning of a non-application is found by evaluating its components.

Combinators and the lambda-calculus form a basis for McCarthy's LISP [30]. The atoms of LISP are character strings (DOG, CAT, 1, CAR, CDR, QUOTE, SURPRISE), and the sole constructor is pair formation (A . B); list notation is an abbreviation for pair notation in commonly occurring LISP expressions; e.g.,

```
(x) = (x . NIL)
(x1 x2 x3 x4) = (x1 . (x2 . (x3 . (x4 . NIL))))
(x1 x2 x3 . x4) = (x1 . (x2 . (x3 . x4)))
```

LISP evaluates expressions as follows:

- If the expression is an atom, the expression associated with it on an auxiliary list is evaluated.

- If the expression is (f . x), and f is an atom, one of three cases holds.
    1. f represents a fundamental LISP combinator.[8] The combinator executes on the evaluation of some portion of x.

---

[8]The atoms representing fundamental LISP combinators are ATOM, EQ, CAR, CDR, and CONS.

2. f = QUOTE. (f . x) evaluates to the first element of x.

3. In any other case, evaluation executes on the pair of the expression associated (via LABEL, see item 1 in the next point) with f and the evaluation list from the elements of x.

- If the expression is (f . x), and f is not an atom, there are two cases.
  1. f = (LABEL g h). Evaluation continues with (h . x) as the expression, and the association of g and f attached to the auxiliary list. This is LISP's mechanism for recursion.

  2. f = (LAMBDA y g). Evaluation executes on g with the elements of y associated (via the auxilliary list) with the evaluations of the elements of x. This is LISP's implementation of lambda-expressions.

- Otherwise, evaluation is undefined.

Landin's applicative expressions [26] explore (among other things) the use of the lambda-calculus and the combinator Y (Yx == xYx) to define programs. Y is the basis for recursion; the factorial function

$$f(n) = \text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1)$$

becomes

$$Y(\lambda f.\lambda n.\{\text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1)\}) .$$

Applicative expressions anticipate CALs; Landin defines their structure through a precursor to constructor syntax[9], the meaning of an expression derives from the meanings of its components, meanings are constants, and the evaluation mechanism (SECD machine) uses an explicit ap-constructor. Landin's applicative expressions are not CALs because the meaning of a lambda-expression is not an applicative expression.

Backus introduced closed applicative languages in 1973 [5], and elaborated on them in 1978 [6] and 1981 [7].

- The 1973 paper presents CALs as the innermost members of a four-level containment hierarchy,

```
[programming languages
    [complete languages
        [applicative languages
            [CALs]]]]
```

  delineates the six formal properties given in Chapter 1, and defines two CALs related to FFP and one related to lambda-calculus.

[9]Expressions are either an identifier or a construction of independent components, and require no detailed syntax.

- The 1978 paper specifies FFP, provides a mechanism underlying the definition of functions, describes a simple system of tranformations between functions, and defines Applicative State Transition Systems (AST systems) as a CAL-based complete computing system. AST systems execute FFP expressions with respect to an internal state including function definitions, and may modify this state after a successfully completed execution. AST systems are weak in at least two respects:

    1. Computations do not persist over a stream of inputs; instead, a state containing possible computations does.

    2. There is no mechanism for communication between modules.

- The 1981 paper continues the study of function transformation begun in 1978.

2.2 Similar research

Friedman and Wise [18, 19], and Henderson and Morris [22] independently develop more-defined LISPs. Both of these systems delay the evaluation of an expression until the result of that evaluation is required by an entire computation, and retain the resulting value as long as a part of the computation might require it; e.g., standard LISP evaluates

```
(CAR (CDR (CDR
  ((LABEL FF (LAMBDA (X)
     (CONS X (FF (PLUS X (QUOTE 1)))))) (QUOTE 1)))))
```

forever[10], but both more-defined LISPs produce 3 as their result. The principal difference between the two systems is that Henderson and Morris provide a complete new evaluation mechanism while Friedman and Wise concentrate on suspended evaluation of CONS (the construction operation). Suspended evaluation in LISP is similar to partial evaluation in more-defined FFP (section 3.4).

Friedman and Wise [20, 21] address the issue of indeterminacy in LISP through the new constructor FRONS. FRONS constructs a list which suspends both the evaluation of elements and the definition of their order; e.g., (FRONS x (FRONS y (FRONS z NIL))) produces a pseudo-list[11] {x y z} whose elements are x, y, and z in no special order. A FRONS structure gains order as selectors require it, and the structure retains this order through all subsequent probes;

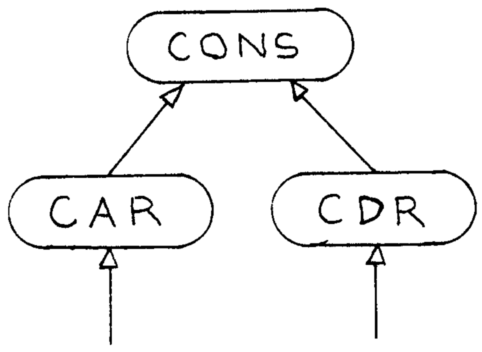
---

[10](LAMBDA (X) (CAR (CDR (CDR X)))) denotes the selector of the third element in a list, and (LABEL FF (LAMBDA (X) (CONS X (FF (PLUS X (QUOTE 1)))))) represents a function which infinitely computes the list of integer beginning with a seed value.

[11]This pseudo-list is called a multiset.

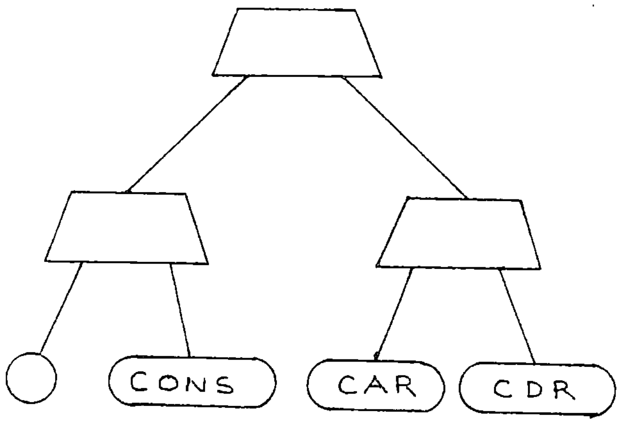
e.g., if any occurence of (FIRST {x y z}) in a computation evaluates to y, then so do all occurrences of (FIRST {x y z}), and all occurences of (REST {x y z}) evaluate to {x z}[12]. This retention of order resembles the pairing of intrinsic determinacy with extrinsic indeterminacy in CFFP (section 4.4).

Keller, Lindstrom, and Patil [24, 25] propose an architecture to execute a language based on more-defined LISP. Their language (FGL) is a graphical version of LISP with operations residing at the nodes, and their architecture is a tree structure with computation and I/O processors at the leaves and communication units at the internal nodes; e.g., FGL depicts (LAMBDA (X Y) (CONS (CAR X) (CDR Y))) as



and the tree aarchitecture may distribute the expression's execution on some unspecified argument as

[12]Friedman and Wise use FIRST instead of CAR and REST instead of CDR.



FGL is a valuable tool in analyzing LISP-like languages, and the tree architecture can execute a wide variety of loosely related languages.

Berkling [8, 9] describes a variation on the lambda-calculus and a conventional stack-oriented implementation of it. Berkling's language uses three explicit application constructors to indicate distinct computation rules; if (f:x) is an application with lambda-expression f, the three computation rules are:[13]

1. $\underline{m}(f{:}x) == \underline{m}\{\{\underline{r}f\}x\}$

2. $\underline{m}(f{:}x) == \underline{m}\{\{\underline{r}f\}\{\underline{m}x\}\}$

3. $\underline{m}(f{:}x) == \underline{m}\{\{\underline{r}\{\underline{m}f\}\}x\}$

Berkling's language also eliminates the need for a set of

[13]The description of computation rules uses the notation of section 1.3.

bound variables by using a correspondence constructor on a single bound variable; if w is the variable, and / is the correspondence constuctor, both $\lambda a \lambda bab$ and $\lambda x \lambda yxy$ become $\lambda\lambda$/ww. This use of a correspondence constructor parallels the technique for finding the association between holes and dapps in CFFP (section 4.4).

2.3 Other related research

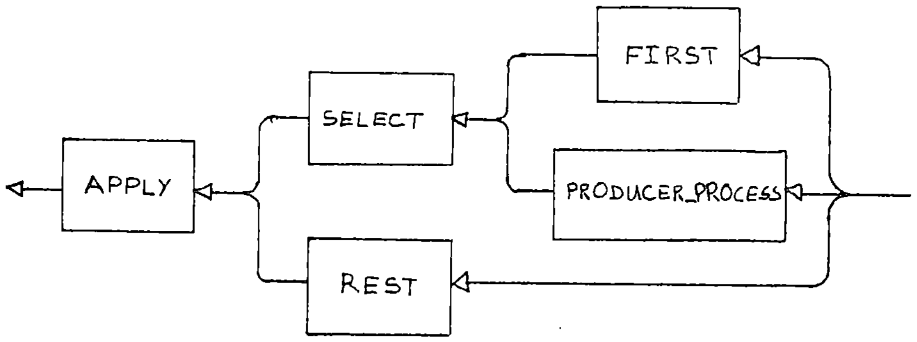
Dataflow languages [3, 16] address many of the same issues as CALs. The fundamental model of these languages is a directed graph with operators at the nodes; computations occur as tokens representing instances of data flow along the arcs, and trigger execution of the operators by arriving at nodes. Dataflow streams [2, 36] are orderings over tokens, and inspired the notion of streams in more-defined FFP (section 3.3). Dataflow resource managers [2] provide controlled access to shared information, and strongly resemble CFFP resource supervisors (section 4.7).

Tonge and Cowan [34] examine the ways that two or more processes might cooperate, and propose a set of process constuctors similar to those of section 4.1. The two sets diverge at two points:

1. Tonge and Cowan include a case-like alternative constuction which routes its input through one of its component sub-processes based on a selector. The selector is the first element of the input.

CFFP (section 4.4) achieves a similar result through a combination of serial and parallel construction.[14]

S(P(S(P(first,producer_process),select),rest),apply)



2. They exclude cycles, and achieve cyclic results through a form of recursion. CFFP retains cycles to deal with issues developed in chapter 4.

Lucid [4] is an assertion-oriented programming language. In Lucid, equality assertions replace assignment statements, programs are an unordered set of such assertions, and the fundamental data concept is an infinite stream (history of values in a variable). These characteristics facilitate a system for proving properties

---

[14]S denotes serial process construction, and P denotes parallel construction.

of Lucid programs, and the success of streams in such a system suggests their value as a programming tool.

2.4 CAL architectures

Mago [27, 28] proposes a tree-structured architecture for an FFP machine. The leaves of the tree are processors which hold the representation of an expression on a symbol-by-symbol basis; each leaf is connected to its parent, and to its left and right neighbor leaves. The internal nodes are also processors, and form a full binary tree over the leaves. The machine executes by repeatedly transforming all innermost applications until none remain. Leaves perform the actual transformations; the internal nodes partition the expression into its subexpressions, determine the innermost applications, and coordinate the actions of the leaves.

Frank [17] addresses a potential problem in the Mago machine: manipulating large data blocks often entails significant, unnecessary processsing overhead. The Frank machine avoids this problem by representing large static data blocks with a pointer to a secondary memory. In this scheme, the machine often only manipulates the pointer rather than the entire data block. Frank shows that machines employing such a secondary memory mechanism are correct implementations of CALs, and that such mechanisms result in improved performance for some programs.

CHAPTER 3

Applications, processes, and computing

A computation is a self-contained activity, but an activity in a computing system (e.g., an operating system) may depend on the states of other independent elements within the overall system. These elements may be resources (card readers, line printers, files, etc.), other activities, or the system itself. Since the states of these elements cannot be predicted in advance, a computing activity is inately history-sensitive, and requires some mechanism for communicating with the other elements of its environment. Operators in such a system should be non-strict since the activities they represent generally depend only on the instantaneous presence of a small part of their input. Computing activities are called processes.

3.1 Processes

A process is the basic activity in computing systems. It executes computations which may depend on external factors. A process may also represent resources or the entire computing system.

Processes represent resources by acting as <u>resource managers</u>. Line printers and files are typical resources.

- The manager of a line printer might accept tagged output requests for printing. The tags indicate a process of origin. When the line printer is quiescent, its manager selects a request and commences a print-job (activating the printer). While the printer is active, the output manager sends requests which bear the tag of the current job to the printer, and holds all other requests for future processing.

- A file manager accepts requests to read from or write to a file. Any number of readers may access a file concurrently, but writers must have exclusive access. The file manager must implement a scheduling algorithm which enforces these restrictions.

Independent processes form a computing system when they cooperate in some activity.

- If an input manager, an output manager, and some executing user-programs interact, they form a system.

- A file manager forms a system with those processes which wish to read from or write to the file.

Cooperation between various activities implies some form of communication among them. A process can be a computing system by containing sub-processes and providing them with a communication mechanism.

3.2 Applications which compute

Processes are the activities in computing systems, and applications are the activities in CALs. With this parallel, using applications as processes seems natural; however, unmodified applications cannot serve as processes for two reasons:

1. Applications view computations as isolated events. With this outlook, they cannot be history-sensitive.

2. Applications are completely self-contained. This restricts an application to communication with only those applications which are "above" it (i.e., those which syntactically contain it).

History-sensitive, fully-communicating applications could serve as processes in a CAL-like computing system.

If applications required only enough information to begin a computation, they could execute on a stream of inputs. This is exactly the form of history-sensitive behavior required of processes. Allowing applications to execute on less than completely specified arguments requires changes in the meaning function of a CAL. Section 3.4 presents such a modification to $\underline{m}$ in FFP.

As CALs are currently defined, an application communicates only to those which contain it. This results in purely progeny-to-parent communication, and coordinate communication is impossible. Such directional constraints do not exist if we introduce a new construct which represents applications inside themselves. Chapter 4 adds this construct and its associated semantics to the more-defined semantics for FFP developed in section 3.4.

3.3 Streams

History-sensitive behavior requires a construction representing the history of inputs to and outputs from a function. Streams may serve this purpose just as they do in dataflow languages [3, 2, 36]. Dataflow languages are closely related to CALs; most programs written in one class are easily translated into the other, and their descriptions of computation correspond.

A stream is a potentially infinite sequence of information packets with a distinct termination for any finite case. A packet (information packet) is a structured or unstructured value which is the input to or output from a computation. Packets must be unique parts of a particular stream, and their order in an argument to a function can define a history of inputs to that function. Some dataflow systems express streams as an ordering of packets on an input or output line.

I define FFP-streams through three constructions:

1. The principal expression of a stream is a pair

<x,y>, where x represents the first packet in the stream, and y (a stream) denotes the rest of the stream.

2. The null atom ϕ expresses the distinct termination of a finite stream; e.g., <A,<B,<C,ϕ>>> is a three element stream.

3. A stream generator is represented by an application. If GEN acting on an integer produces the pairing of that integer with the application of GEN to the integer's successor (e.g., (GEN:4) -->> <4,(GEN:5)>), then <1,<2,(GEN:3)>> is one possible denotation for the stream of positive integers.

The expression

<z1,<z2,(f:<y1,<y2,<y3,(g:<x1,<x2,ϕ>>)>>>)>>

depicts a history-sensitive computation. f and g represent history-sensitive functions. g is a stream generating function, acts on a finite stream of inputs (of which x1 and x2 remain), and sends its stream of results (represented by y1, y2, and y3) to f. f is also a stream generating function and acts on g's output to produce the ultimate stream of outputs (z1 and z2).

3.4 A more-defined semantics

We want an application to treat a stream as a constant argument whenever that stream's first element is a constant, but the meaning function does not allow such behavior. For example, if we formally define GEN by

```
DEF GEN == [ID,GEN*ADD*[ID,'1]] ,
```

and form the application

```
(ADD*[ADD*[1,1*2],1*2*2]*GEN:1)
```

to add the first three positive integers, evaluation never terminates even though a result of 6 is reasonable. This situation results from trying to totally evaluate (GEN:1), which represents an infinite structure.

Friedman and Wise [18], and Henderson and Morris [22] address a similar situation in LISP through evaluation-by-demand schemes. In such schemes, the semantic mechanism begins at the outermost level of an expression, and evaluates inner sub-expressions only to the extent required for determining the meaning of that outermost level. That is, the above example (GEN:1) would evaluate only to

```
<1,<2,<3,(GEN*ADD*[ID,'1]:3)>>> .
```

We define a partial meaning function $\underline{p}$ that supports demand-evaluation in FFP. Given any expression, $\underline{p}$ returns the least-evaluated non-application derived from it. For example,

$\underline{p}$(GEN:1) = <(ID:1),(GEN*ADD*[ID,'1]:1)>

since the right-hand side expresses the form of the result without evaluating either of the internal applications, and

$\underline{p}$(1*GEN:1) = 1

since any less-evaluated result is still an application. The partial meaning of any non-application is itself.

We can define the meaning function to use $\underline{p}$ to evaluate applications.

```
mx == x∈A => x;
      x = <x1,...,xn> => <mx1,...,mxn>;
      x = (f:z) => m{px};
      ⊥
```

$\underline{p}$ returns a non-application, and $\underline{m}$ continues its evaluation if necessary.

One specification of $\underline{p}$ is

```
px == x∈A => x;
      x = <x1,...,xn> => x;
      x = (f:z) => {f∈A => {df=# => p{{rf}z};
                            df=g => p(g:z)};
                    f = <f1,...,fj> => p(f1:<f,z>);
                    p(pf:z)};
      ⊥
```

This $\underline{p}$ does not evaluate the argument parts of applications; instead, it passes this responsibility to the primitive functions. These revised functions[15] invoke $\underline{p}$ to produce the required form of argument; e.g.,

---

[15]Appendix I provides a list of primitive functions.

```
{r1}x == x = <x1,...,xn> => x1;
         x = (f:z) => {r1}{px};
         ⊥
```

I call the evaluation mechanism based on this p the "interacting semantics". Under it, (1:(GEN:1)) evaluates as shown in figure 5.

```
m(1:(GEN:1)) = m{p(1:(GEN:1))}
  = m{p{{r1}(GEN:1)}}
  = m{p{{r1}{p(GEN:1)}}}
  = m{p{{r1}{p([ID,GEN*ADD*[ID,'1]]:1)}}}
  = m{p{{r1}{p(CONS:<[ID,GEN*ADD*[ID,'1]],1>)}}}
  = m{p{{r1}{p{{rCONS}<[ID,GEN*ADD*[ID,'1]],1>}}}}
  = m{p{{r1}{p<(ID:1),(GEN*ADD*[ID,'1]:1)>}}}
  = m{p{{r1}<(ID:1),(GEN*ADD*[ID,'1]:1)>}}
  = m{p(ID:1)}
  = m{p{{rID}1}}
  = m{p1}
  = m1
  = 1
```

Figure 5. An evaluation by the interacting semantics

Another version of partial meaning partitions the standard primitive atoms [5, 6] into six sets[16] according to their least-defined useful arguments. Figure 6 enumerates the partition and its corresponding sets of useful arguments. The Pi define the partition, and the Ai describe the corresponding useful arguments.

The new version of partial meaning [figure 8] uses the

[16]Other sets of primitive atoms could require different partitions.

```
P1 = {AA,AL,AR,CN,EQAT,FST1,INSERT,AND,OR,
      ADD,SUB,MUL,DIV,LT,LE,GT,GE}
A1 = {<x,y>| x,y≠(f:z)}

P2 = {TRANS}
A2 = {<x1,...,xn>| xi=<y1,...,yj> or xi=φ}

P3 = {APNDL,DISTL}
A3 = {<x,y>| y≠(f:z)}

P4 = {APNDR,DISTR,COMP,CONS,CONST,IF,WHILE,BU}
A4 = {<x,y>| x≠(f:z)}

P5 = {APPLY,ATOM,LENGTH,NOT,NULL,REVERSE,ROTL,
      ROTR,SEPL,SEPR,TL,TLR,s,sR,T,F}
A5 = {x| x≠(f:z)}

P6 = {DBL,ID}
A6 = E
```

Figure 6. A partition of primitive atoms

partition to determine an appropriate depth of evaluation. This "partitioned semantics" requires no interaction between p and the primitive functions. Figure 7 evaluates (1:(GEN:1)) using the new p.

```
m(1:(GEN:1)) = m{p(1:(GEN:1))}
  = m{p(1:p(GEN:1))}
  = m{p(1:p([ID,GEN*ADD*[ID,'1]]:1))}
  = m{p(1:p(CONS:<[ID,GEN*ADD*[ID,'1]],1>))}
  = m{p(1:p{{rCONS:<[ID,GEN*ADD*[ID,'1]],1>}>}
  = m{p(1:p<(ID:1),(GEN*ADD*[ID,'1]:1)>)}
  = m{p{{r1}<(ID:1),(GEN*ADD*[ID,'1]:1)>}}
  = m{p(ID:1)}
  = m{p{{rID}1}}
  = m{p1}
  = m1
  = 1
```

Figure 7. Evaluation using partitioned partial meaning

The two versions of partial meaning differ only in minor details. The interacting p blurs the distinction

```
px == x∈A => x;
      x = <x1,...,xn> => x;
      x = (f:z) =>
          {f∈A =>
           {df=# =>
            {f∈P1 => {z∈A1 => p{{rf}z};
                      z=<z1,z2> => p(f:<pz1,pz2>);
                      p(f:pz)};
             f∈P2 => {z∈A2 => p{{rf}z};
                      z=<z1,...,(g:y),...,zn>
                       => p(f:<pz1,...,pzn>);
                      p(f:pz)};
             f∈P3 => {z∈A3 => p{{rf}z};
                      z=<u,(g:y)> => p(f:<u,p(g:y)>);
                      p(f:pz)};
             f∈P4 => {z∈A4 => p{{rf}z};
                      z=<(g:y),u> => p(f:<p(g:y),u>);
                      p(f:pz)};
             f∈P5 => {z∈A5 => p{{rf}z};
                      p(f:pz)};
             {{rf}z};
            df=g => p(g:z)};
          f = <f1,...,fn> => p(f1:<f,z>);
          p(pf:z)};
      ⊥
```

Figure 8. Partitioned partial meaning

between primitive represented functions and the underlying semantic functions, and the partitioned p introduces a very weak typing of functions.[17] Both compute the same results in the same history-sensitive manner, and choosing between them is a matter of momentary convenience. We will call semantics based on the revised m and either p more-defined.

The more-defined semantics change the nature of five

[17]This typing does not introduce a hierarchy of functions, and thus does not violate the spirit of the single-type functions property.

primitive functions. EQ cannot represent a primitive function since an appropriate depth of evaluation cannot be predetermined. T can result only from a totally evaluated argument, but F can reasonably result from less than total evaluation. IF, WHILE, T, and F should represent more asynchronous and evocative functions.

EQ's function becomes

```
DEF EQ == <IF,AND*@ATOM,EQAT,
          <IF,OR*@ATOM,'F,
          <IF,EQAT*@LENGTH,/AND*@EQ*TRANS,'F>>>
```

This new definition of EQ produces results whenever the old one does so, and also produces some results when the original primitive definition does not. With the more-defined semantics and new EQ,

```
(EQ:<(TPL:1),(2PL:1)>)
```

returns F when

```
DEF TPL == [TPL,TPL,TPL]
DEF 2PL == [2PL,2PL]   ,
```

but evaluation under basic FFP never terminates. Both TPL and 2PL recurse infinitely, but under one execution of partial meaning (TPL:1) produces

```
<(TPL:1),(TPL:1),(TPL:1)>
```

and (2PL:1) produces

```
<(2PL:1),(2PL:1)> .
```

With this structural information, application of the new EQ returns F based on a difference in length between two sequences, rather than insisting that (TPL:1) and (2PL:1) be fully evaluated.

Under the more-defined semantics, primitive functions can use applications more freely in their definitions, and an application can indicate potential computations as well as essential ones. IF, WHILE, T, and F become

```
{rIF}x == x = <<z,p,f,g>,y>
                => ((p:y):<(f:y),(g:y)>);
              ⊥

{rWHILE}x == x = <<z,p,f>,y>
                   => ((p:y):<(<WHILE,p,f>:(f:y)),y>);
                 ⊥

{rT}x == x = <y,z> => y; ⊥

{rF}x == x = <y,z> => z; ⊥
```

under the partitioned semantics.[18] These new functions denote the structure of their computations more clearly and concisely than the original functions, even though an application using either version produces the same ultimate result.

The revised transition function $\underline{t}$ transforms a broader range of applications under the more-defined semantics. Instead of replacing innermost applications, $\underline{t}$ replaces

[18]See Appendix I for the interacting semantics.

applications whose function and argument parts are in corresponding sets under the partitioned semantics. t could substitute for any of the applications in

(1:<(ADD:<4,5>),(APNDL:<(ID:9),ϕ>)>) ,

with the outermost as a direct route to the meaning[19] In general, defined applications denotes the class for which t can substitute.

3.5 Sieve of Eratosthenes

A prime number generator based on the Sieve of Eratosthenes demonstrates the history-sensitive behavior now possessed by applications. Figure 9 defines the necessary functions:

```
DEF PRIMES == PASS*[ID,SIEVE*GEN*'2]

DEF PASS == <IF,EQAT*[1,'0],'ϕ,
                 [1*2,PASS*[SUB*[1,'1],2*2]]>

DEF SIEVE == [1,SIEVE*DIVBY*[[1,1],2]]

DEF DIVBY == <IF,EQAT*[2*1,1*2],DIVBY*[1,2*2],
              <IF,LT*[2*1,1*2],DIVBY*[[1*1,ADD*1],2],
                 [1*2,DIVBY*[1,2*2]]>>

DEF GEN == [ID,GEN*ADD*[ID,'1]]
```

Figure 9. A prime number generator

- (PRIMES:n) generates a stream consisting of the first n prime numbers.

[19]The semantics do not specify this route, but do allow it.

- (PASS:<n,x>) passes the first n elements of stream x.

- (SIEVE:x) passes those elements of stream x which are not divisible by their predecessors. It expects an infinite input stream.

- (DIVBY:<<n,n>,x>) passes those elements of stream x which are not divisible by n.

- (GEN:2) generates an infinite stream of numbers begining with 2.

$\underline{\mathrm{m}}$(PRIMES:5) = <2<,3,<5,<7,<11,ϕ>>>>>, and (SIEVE*GEN:2) would try to produce the entire infinite set of primes (until the physical limits of the executing machine were reached). PASS, SIEVE, and DIVBY act on a sequence of inputs rather than expecting totally evaluated arguments; in fact, the stream produced by SIEVE would terminate with $\perp$ if its input stream were finite. Only history-sensitive functions can manifest such behavior.

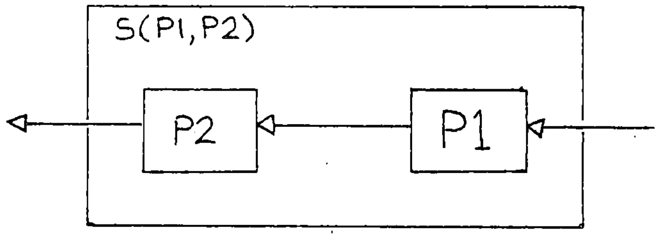
CHAPTER 4

Complete computing systems

4.1 Process interactions

Two or more processes may cooperate in performing some operation. In doing so, they form a new composite process. The following three structural forms specify the nature of this cooperation.

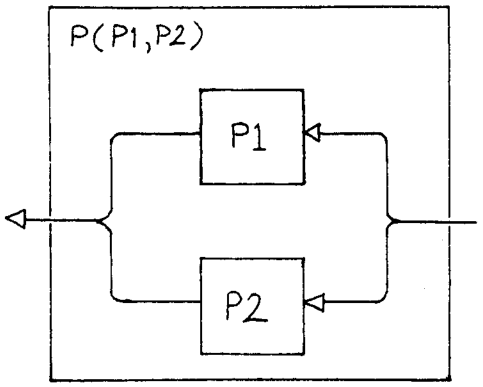
1. Serial execution. Two processes, P1 and P2, form a composite process, S(P1,P2), such that the composite's input is the input to P1, the composite's output is that of P2, and the output of P1 is the input of P2.



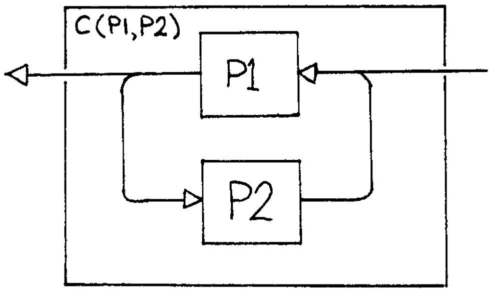
2. Parallel execution. P1 and P2 act independently on a common input, and combine[20] their outputs

[20]Any information-preserving combination may be used since a following operator can produce an alternate combination.

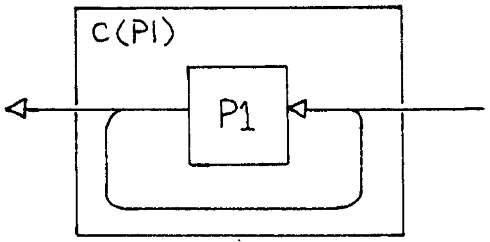
into a common output. This forms P(Pl,P2).



Cyclic execution. Pl produces a stream of outputs which are input to P2, and P2's output forms part of the input to Pl. The composite output is the output of Pl. This creates C(Pl,P2).



P2 may be the identity process, and we will often omit it and write C(Pl).



These constructions easily expand to include an arbitrary number of processes; e.g.,

```
S(P1,P2,P3) == S(P1,S(P2,P3))
P(P1,P2,P3) == P(P1,P(P2,P3))
C(P1,P2,P3) == C(P1,S(P2,P3))
```

Tonge and Cowan [34] also examine the ways that two or more processes might cooperate; they propose a slightly different set of constuctors as described in section 2.3.

More defined FFP easily expresses serial and parallel executions. Function composition indicates the same flow of information from P1 to P2 as does the serial process structure,

```
S(P1,P2) == P2*P1
```

and functional constuction describes parallel execution.

```
P(P1,P2) == [P1,P2]
```

Sections 4.3 and 4.4 address cyclic execution.

4.2 Indeterminacy

Processes in a real computing system (e.g., an operating system [32]) interact in fundamentally indeterminate ways. A file manager FM might interact with three other processes (A B C) by granting requests to read from or write to a file; typically the processes might produce the following input streams to FM.

```
A:  <read,<read,<write,(fA:sA)>>>
B:  <write,(fB:sB)>
```

```
C:  <read,<write,<read,<write,(fC:sC)>>>>
```

fp is the current action, and sp is the current state of p. The first input to FM will be either the leading read from A, the write from B, or the initial read from C; but we cannot determine which until FM receives it. Thus, A, B and C interact to form an indeterminate input to FM.

A model of a real computing system must provide a mechanism for combining the outputs of processes in the above indeterminate way. Indeterminate merge, denoted ⊠, is an operator which performs the required combination. ⊠ operates on two (or more) streams of values by arbitrarily selecting the first element of either argument as the first element of its output, and then recurring on its other inputs and the rest of the selected parameter. The primitive operator NEXT is the basis for ⊠ in more-defined FFP.

```
{rNEXT}x == x = ϕ => ϕ;
            x = <x1,...,xn>
              => {{∃xi:pxi=<y1,y2>} => i;
                  {∀xi:pxi=ϕ} => ϕ; ⊥}; ⊥
```

Since there may be many possible i's to return, NEXT is indeterminate. We may now define ⊠ in a set of definitions.

```
DEF ⊠ == <IF, NULL*1, 'ϕ,
          [1*APPLY,⊠*NONULLS*REST]>*[NEXT,ID]

DEF REST == <IF, EQ*[1,'1], APNDL*[2*1*2,TL*2],
                 APNDL*[1*2,REST*[PRED*1,TL*2]]>
```

```
DEF PRED == SUB*[ID,'1]

DEF NONULLS == <IF, NULL, 'ϕ,
                <IF, NULL*1, NONULLS*TL,
                     APNDL*[1,NONULLS*TL]>>
```

NEXT returns an index, and ⊠ uses the first element at that index as the next output element. In addition, ⊠ recurs on its input with the indexed value replaced by the second element found there, and all null values removed.

At first glance, the above definition for indeterminate merge may appear erroneous; if the expression indexed by NEXT is indeterminate, the output from ⊠ could be inconsistent with the recursion. That is

(⊠:<(⊠:<<1,<2,ϕ>>,<3,ϕ>>),<4,ϕ>>)

might produce

<1,<4,<1,<2,ϕ>>>>

even though this stream is not a legal possibility. If an expression x in ([f,g]:x) is indeterminate, we must assert that x in (f:x) has the same value as x in (g:x). This form of determinacy is assured if all expressions and functions are intrinsically determinate, and all indeterminacy is extrinsic. Under this restriction, all <u>copies</u> of the same expression have the same meaning, and represent the same determinate function. All apparent indeterminacy resolves once a meaning or action is determined for any copy. Friedman and Wise [20, 21] use a notion similar to intrinsic determinacy in selecting elements from a multiset.

Since indeterminate merge is a possible operation (and hence, a possible process), indeterminate combination of processes I(P1,P2) is not a new construction. Instead, I(P1,P2) becomes a composite of serial and parallel executions.

I(P1,P2) == S(P(P1,P2),⊠) = ⊠*[P1,P2]

4.3 Cycles

Explicit cycles are not required, provided we deal exclusively with continuous functions [29, 33] as processes. Keller [24] describes a transformation of cycles into recursion which is illustrated in figure 10. For a continuous function f with input x, output y, and cycle z, the transformation operates as follows:

- Cut the cycle, and divide the function into two parts, g and h. g produces the result of the original function, and h produces the cycle.

- Recognize that z is just another function h' of x.

- Convert the cycle to a structure written as

$$y = g(x,z) = g(x,h'(x))$$
$$z = h'(x) = h(x,h'(x))$$

In FFP, with representations rather than functions, we may express the transformation as

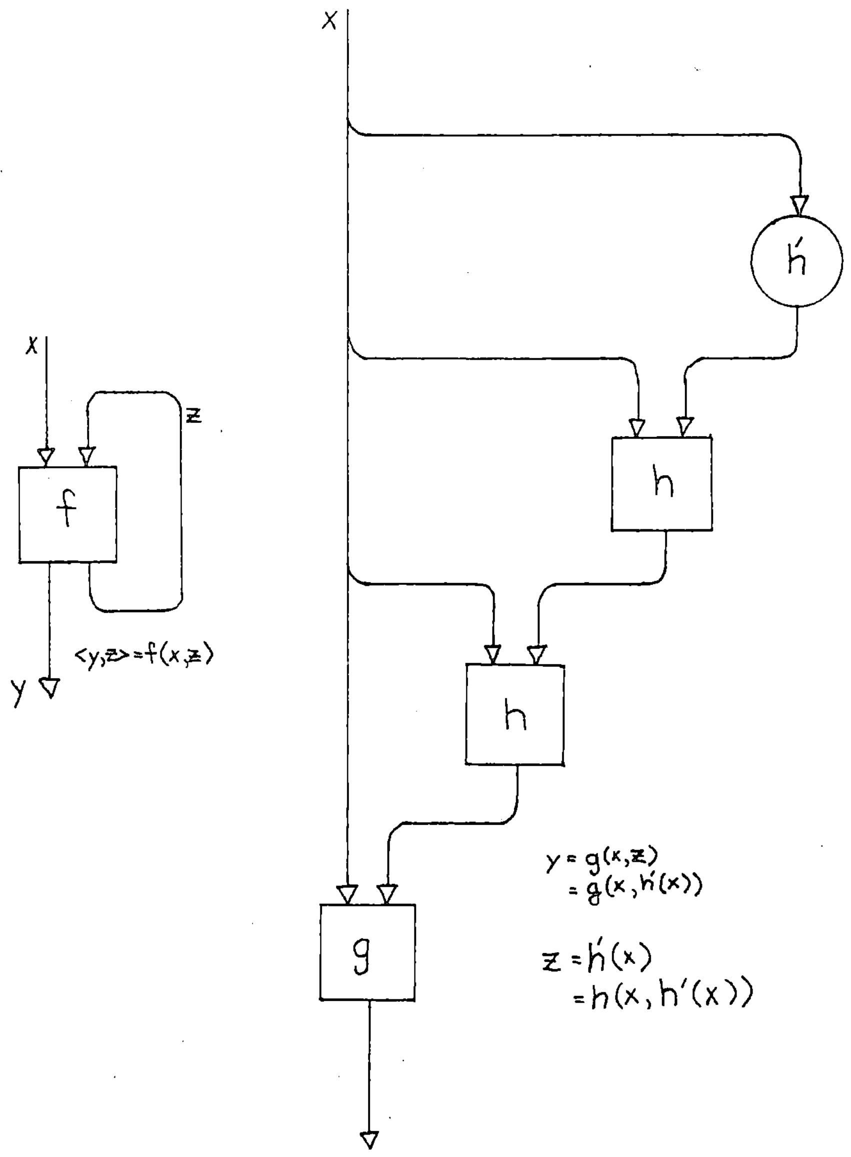


Figure 10. Transformation from Keller

DEF CYCLE == g*[ID,H]

DEF H == h*[ID,H] .

H corresponds to h'.

Unfortunately, Keller's transformation does not succeed for cycles which include non-continuous functions like indeterminate merge. Figure 11 illustrates the transformation of a cycle which employs ⊗.

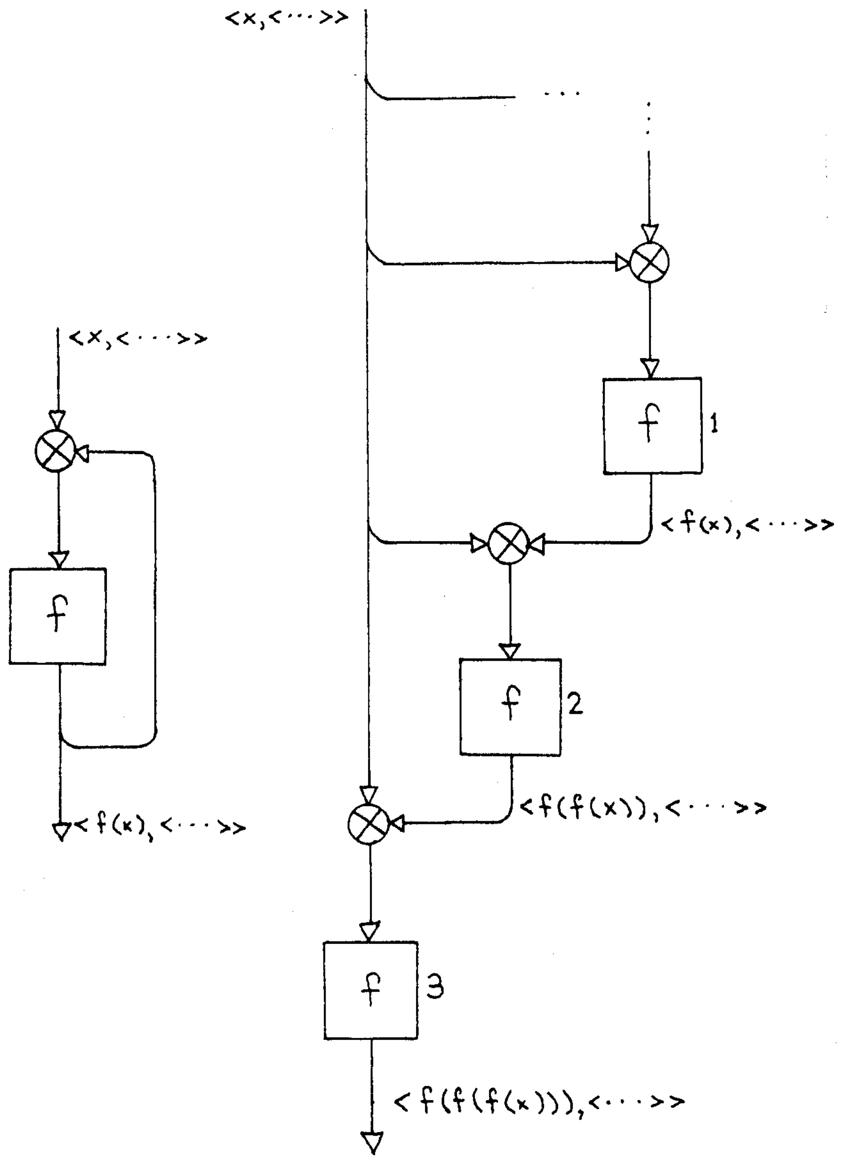


Figure 11. Transformation with indeterminacy

The cycle is determinate even though ⊗ appears, and the first value it produces is f(x). The transformed cycle, however, is indeterminate, and can produce results different from the one produced by the cycle; for example, instance one of f produces f(x), instance two produces f(f(x)), and instance three produces f(f(f(x))) as a possible first value from the transformed cycle.

The inconsistent hehavior does not resolve when we express the transformation in FFP.

```
DEF CYCLE == f*⊠*[ID,CYCLE]
```

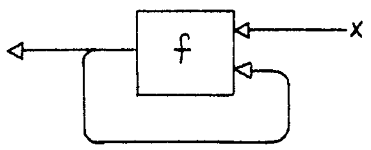
The recursive application of CYCLE will repeatedly instantiate ⊠, and it is the distinct instances of indeterminacy which cause the problem. Intrinsic determinacy does not help either since any two instances of ⊠ will act on distinct expressions rather than copies of a single common expression.

While Keller's transformation does not seem to be the solution for integrating cycles with indeterminacy, we must develop some such solution. Cyclic FFP (CFFP) is that solution.

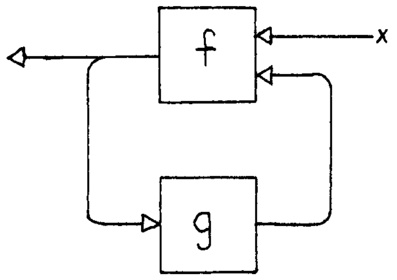
4.4 CFFP

A cycle is a part of a function's result which is also a part of the same function's argument. In FFP, the results of functions are the meanings of applications, and we must express cycles as some representation of an application within itself. Recursion is the conventional approach for achieving this representation, but section 4.3 indicated that it is inappropriate when indeterminacy is considered.

CFFP explicitly represents cycles by pairing two new constructs, distinguished applications (or dapps) and holes. A dapp is a stream-producing application which feeds its results into itself as well as producing them. If f and x are expressions, (f|x) is a dapp. A hole is an expression which copies the output stream of a surrounding dapp. If <ϕ> denotes a hole[21], then (f|<x,<ϕ>>) denotes the cycle

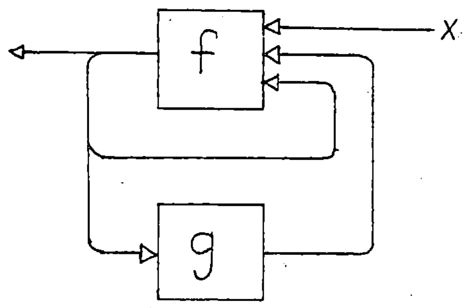


(f|<x,(g:<ϕ>)>) denotes

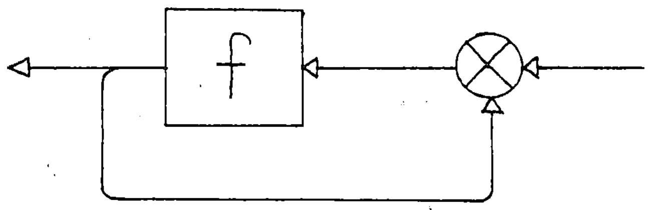


[21]<ϕ> could be any unique symbol.

(f|<x,(g:≺ϕ≻),≺ϕ≻>) denotes



and (f|(⊠:<x,≺ϕ≻>)) denotes



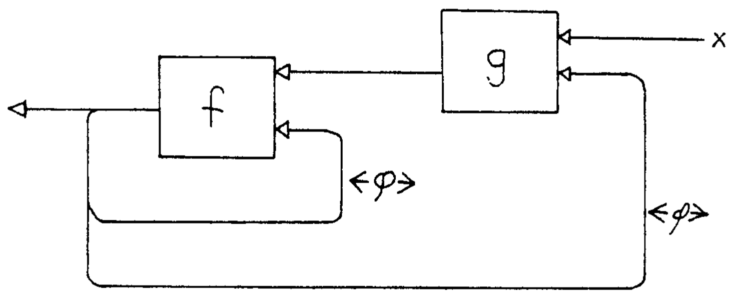
CFFP uses an infinite set of holes $\mathbb{H}$

$$\mathbb{H} == \{≺ϕ≻\} \cup \{≺h≻ \mid h \in \mathbb{H}\}$$

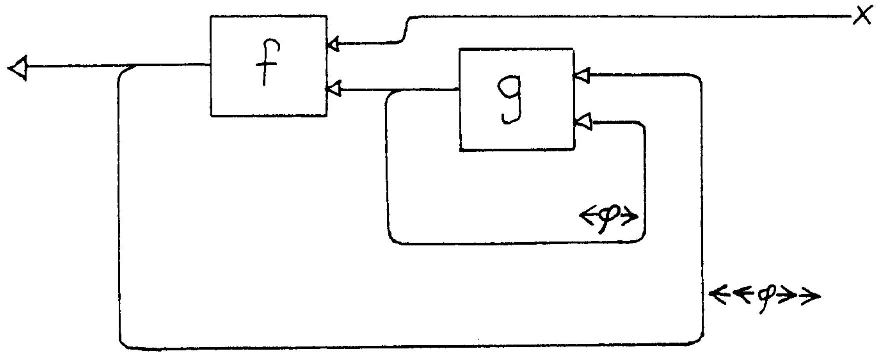
in representing nested cycles such as those depicted in figure 12. A hole indicates its unique dapp by the nesting level of ϕ:

- ≺ϕ≻ refers to the immediately surrounding dapp.

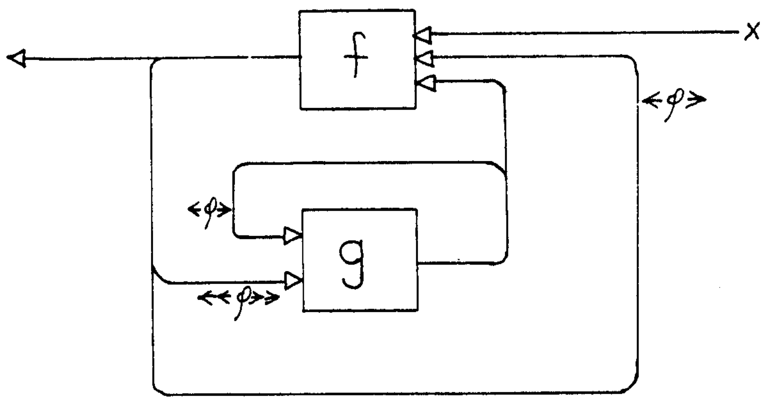
- ≺h≻ (h≠ϕ) refers to that dapp which immediately



(f|<(g:<x,≺φ≻>),≺φ≻>)



(f|<x,(g|<≺≺φ≻≻,≺φ≻>)>)



(f|<x,≺φ≻,(g|<≺φ≻,≺≺φ≻≻>)>)

Figure 12. Nested cycles

surrounds the dapp to which h refers.

The formal specification of CFFP requires two additional constructs:

1. x̊ indicates a hole which references no dapp. The semantic functions (14 through 20) form instances of x̊, and their descriptions explain the circumstances which produce x̊.

2. $\underline{w}$ represents ill-formed. The semantic functions instantiate $\underline{w}$ as (a) the meaning of a hole[22], (b) the meaning or partial meaning of x̊, or (c) the partial meaning (which then becomes the meaning) of a dapp which does not produce a stream of well-formed packets. If any constructed expression contains $\underline{w}$, the entire expression denotes $\underline{w}$ (i.e., $\forall k \in \underline{K}, ei \in \underline{E}$: $k[e1,\ldots,\underline{w},\ldots,en] == \underline{w}$).

The semantic functions also use the set of well-formed dapps $\underline{Z}$ as defined in figure 13. If (f|x) is well-formed, it produces a stream, the packets in that stream all have well-formed meanings[23], and either the stream properly

---

[22]A hole cannot have a meaning since any well-formed dapp either produces an infinite stream, or becomes ϕ (in which case the hole no longer exists).

[23]If an expression has a well-formed meaning, it does not denote $\underline{w}$.

```
Z == {(f|x)| p(f|x)=<y,z>,y∈Y,z∈Z'}
             U {(f|x)| p(f|x)=ϕ}

Z' == {<y,z>| y∈Y,z∈Z'} U Z U {ϕ}

Y == {y| my≠w}
```

Figure 13. Well-formed dapps

terminates in ϕ, or there always exists a well-formed dapp which continues to generate the stream. m and p are the meaning and partial meaning functions for CFFP expressions as defined in the following paragraphs.

Figures 14 through 20 define the semantics of CFFP by adding dapps and holes to the more-defined semantics of section 3.4.

```
mx == x∈A => x;
      x = <x1,...,xn> => <mx1,...,mxn>;
      x = (f:z) => m{px};
      x = (f|z) => m{px};
      x = ⊥ => ⊥;
      w
```

Figure 14. Meaning function

```
px == x∈A => x;
      x = <x1,...,xn> => x;
      x = (f:z) => {f∈A => {df=# => p{{rf}z};
                            df=g => p(g:z)};
                    f = <f1,...,fn> => p(f1:<f,z>);
                    f = (g:y) => p(pf:z);
                    f = (g|y) => p(pf:z);
                    f∈H => p(pf:z);
                    f = ⊥ => ⊥;
                    w}
      x = (f|z)∈Z => y{p(f:z)};
      x∈H => {kx}{p{{θx}x}};
      x = ⊥ => ⊥;
      w
```

Figure 15. Partial meaning function

- The meaning function m (14) adds dapps as another

```
ye == e = ϕ => ϕ;
      e = <u,v> =>
       {v = <x,z> => <{g⋖ϕ⋗}u,y{{{s{{g⋖ϕ⋗}u}}⋖ϕ⋗}v}>;
        v = (f:z) => <{g⋖ϕ⋗}u,({{s{{g⋖ϕ⋗}u}}⋖ϕ⋗}f|
                                {{s{{g⋖ϕ⋗}u}}⋖ϕ⋗}z)>;
        v = ϕ => <{g⋖ϕ⋗}u,ϕ>;
        w};
      w
```

Figure 16. Conversion function

```
{gh}x == x = <x1,...,xn> => <{gh}x1,...,{gh}xn>;
         x = ⋖y⋗ => {x<h => x; x̲̅};
         x = (f:z) => ({gh}f:{gh}z);
         x = (f|z) => ({g⋖h⋗}f|{g⋖h⋗}z);
         x
```

Figure 17. Consistency function

```
{{sz}h}x == x = h => <z,h>;
            x = <x1,...,xn> => <{{sz}h}x1,...,{{sz}h}xn>;
            x = (f:y) => ({{sz}h}f:{{sz}h}y);
            x = (f|y) => ({{sz}⋖h⋗}f|{{sz}⋖h⋗}y);
            x
```

Figure 18. Substitution function

```
{θ⋖h⋗}x == x = x̲̅ => x̲̅;
           h = ϕ =>
             {ǂ(f|z)=glb{(g|y)|x∈(g|y)≠x} => (f|z); x̲̅};
           h∈H => {θh}{{θ⋖ϕ⋗}x}
```

Figure 19. Context function

```
{kh}x == x = (f|z) => {h∈(f|z) => h; w};
         x = <y,z> => <y,{kh}z>;
         w
```

Figure 20. Extraction function

form of application. w, x̲̅, and all holes denote ill-formed:

* An ill-formed expression cannot have a well-formed meaning.

* x̲̅ represents an incomplete cyclic structure.

* A hole cannot receive a terminal value from a well-formed dapp. Either the dapp produces an infinite stream with no terminal value, or it terminates in ϕ (which is an atom with no component subexpressions), and the hole no longer exists.

- The partial meaning function $\underline{p}$ (15) adds dapps and holes at various points.

  * The partial meaning of a well-formed dapp (f|z) is the conversion function $\underline{y}$ applied to the partial meaning of the application (f:z) corresponding to the dapp. $\underline{y}$ makes the changes necessary to feed the results of (f:z) into the holes of (f|z).

  * The partial meaning of a hole only exists as part of the partial meaning of the dapp it references. If <h> is the hole, then {θ<h>}<h> evokes its dapp, and if z is the partial meaning of that dapp, then {$\underline{k}$<h>}z extracts the partial meaning of <h> from z.

  * When an application has a dapp or hole as its function part, $\underline{p}$ treats that application as if it had another application as function

part. p̲ determines the partial meaning of the function-part and recurs on the revised application.

* The primitive functions returned by r̲ treat a hole or a dapp as if it were an application.

- The conversion function y̲ (figure 16) transforms the partial meaning of a stream-producing application (f:z) into the partial meaning of the corresponding dapp (f|z); e.g., with no holes or dapps in x, x', y, or g:

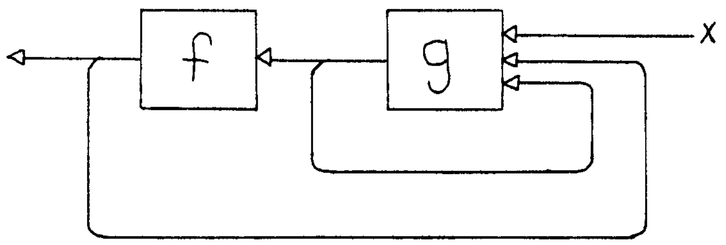
```
p̲(f:<⋖ϕ⋗,x>) = <y,(f:<⋖ϕ⋗,x'>)>
  => y̲{p̲(f:<⋖ϕ⋗,x>)} = <y,(f|<<y,⋖ϕ⋗>,x'>)>

p̲(f:<⋖ϕ⋗,x>) = <⋖ϕ⋗,(f:<⋖ϕ⋗,x'>)>
  => y̲{p̲(f:<⋖ϕ⋗,x>)} = <x̂,(f|<<x̂,⋖ϕ⋗>,x'>)>

p̲(f:<x,⋖ϕ⋗,⋖⋖ϕ⋗⋗>)
      = <<y,⋖⋖ϕ⋗⋗>,(f:<x',⋖ϕ⋗,⋖⋖ϕ⋗⋗>)>
  => y̲{p̲(f:<x,⋖ϕ⋗,⋖⋖ϕ⋗⋗>)}
          = <<y,x̂>,(f|<x',<<y,x̂>,⋖ϕ⋗>,⋖⋖ϕ⋗⋗>)>

p̲(f:<⋖ϕ⋗,(g|<x,⋖ϕ⋗,⋖⋖ϕ⋗⋗>)>)
     = <(g|<x,⋖ϕ⋗,⋖⋖ϕ⋗⋗>),(f:⋖ϕ⋗)>
  => y̲{p̲(f:<⋖ϕ⋗,(g|<x,⋖ϕ⋗,⋖⋖ϕ⋗⋗>)>)}
        = <(g|<x,⋖ϕ⋗,x̂>),(f|<(g|<,⋖ϕ⋗,x̂>),⋖ϕ⋗>)>
```

* For each packet produced, y̲ invokes two additional functions:

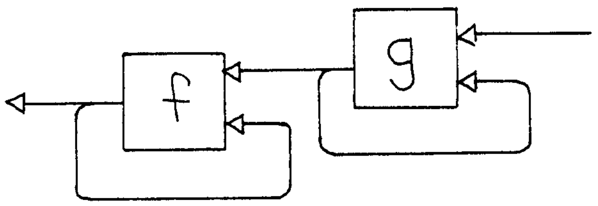
1. The consistency function g̲ (17) converts all free holes[24] into x̂. Such

[24]A hole ⋖h⋗ is free within a packet if the packet does not contain the dapp to which ⋖h⋗ refers.

holes refer to either the dapp which produced them or a dapp which contains that dapp. In the first case, the holes have lost their context, and in the second, they represent a change in the cyclic structure of the overall expression; e.g.,



becomes



This change is disallowed because it originates inside the cycle (at g) and not outside (at f or some expression which contains f and g). The set of holes $\underline{H}$ is ordered by nesting; thus, $\forall x,y \in \underline{H}$: $y < \langle y \rangle$, and $\langle y \rangle \leq \langle x \rangle$ iff $y \leq x$.

2. The substitution function s (18) feeds packets converted by g to holes which reference the producing dapp. {{gz}h}x can be loosely read as substitute z for h in x.[25]

* When y encounters a terminal application, y converts the application to a dapp.

- The context function θ (figure 19) evokes the dapp to which a given hole refers. For the second occurence of $\lessdot\phi\gtrdot$ in $(f|\langle x,\lessdot\phi\gtrdot,(g|\langle y,\lessdot\phi\gtrdot,\lessdot\lessdot\phi\gtrdot\gtrdot\rangle)\rangle)$, $\{\theta\lessdot\phi\gtrdot\}\lessdot\phi\gtrdot$ returns $(g|\langle y,\lessdot\phi\gtrdot,\lessdot\lessdot\phi\gtrdot\gtrdot\rangle)$. The resulting expression is not a copy of the dapp; instead, it is the specific instance of the dapp which contains the given hole. For expressions x and e, $x \in e$ indicates that a particular instance of x is found in a specific instance of e. glb denotes the greatest lower bound ordered by syntactic containment (e.g., $(f|x) \leq (g|z)$ iff $(f|x) \in (g|z)$).

- The extraction function k (figure 20) returns the stream produced at a hole. k does this by copying

[25]This is imprecise because each dapp encountered deepens the nesting of h by one.

each packet produced by the hole's dapp, and replacing that dapp with the given hole. k also checks to see if the hole continues to exist in the dapp, and returns w if it does not.

A CFFP transition function t augments the semantic functions. This t is that of the more-defined semantics with dapps and holes added as additional forms of application. A dapp (f|x) is defined (subject to execution) whenever its corresponding application (f:x) is defined, and a hole is never defined.[26] Defined computations (applications and dapps) seem closely related to live processes, and studies of CFFP-based computing systems may profit from any such relationship.

4.5 Formal properties

Backus' six formal properties for CALs [5] hold for CFFP with minor modifications.

1. Idempotency of meaning. This property holds without modification. A meaningful expression still evaluates to a constant expression, and constant expressions always evaluate to themselves.

[26] Alternatively, a hole may be defined together with the dapp it represents. In this case, a transition on a hole is the same as a transition on the corresponding dapp.

2. _The "anti-quote" property_. This property must allow holes and dapps as non-constant expressions in the same sense as applications. With this in mind, constant expressions still represent themselves, and representing values still requires no special quoting mechanism.

3. _Non-extensionality_. CFFP implies no identity between the meaning of an expression and the function it represents. As such, non-extensionality holds.

4. _Single-type functions_. The details of this property have changed, but the substance remains the same. Functions are now mappings from expressions into expressions, rather than constants into expressions, but they still map from a single common domain onto a single common range.

5. _The extended Church-Rosser property_. This property no longer possesses its original extension since some reduction sequences could try to evaluate a non-terminating sub-expression; however, all terminating sequences of reductions, on a given expression, still yield the same

meaning for it. The extension is replaced by

> Every finite sequence of reductions on a meaningful expression forms the prefix of some terminating reduction sequence.

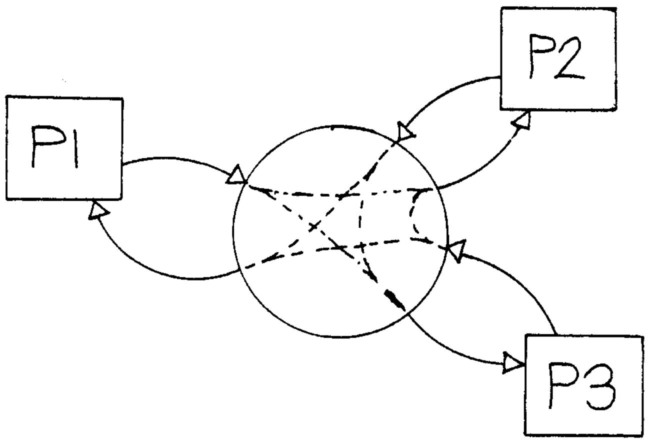
6. _The reduction property_. Some sub-expressions of a meaningful expression may be meaningless (i.e., they may represent non-terminating computations). With this in mind, the reduction property becomes

> If a given meaningful sub-expression is replaced by its meaning, The meaning of the parent expression remains unchanged.

Appendix II formally addresses these properties.

4.6 Interprocess communication

CFFP can describe interprocess communication. Consider a system with three processes and message-oriented communication.



If one process wishes to send a message to another, it produces a pair <d,m> where d is the destination of the message, and m is the message. If f1, f2, and f3 represent the processes, the complete system is

```
(⊠|<(f1:<x1,(GUARD1:←ϕ→)>),
     (f2:<x2,(GUARD2:←ϕ→)>),
     (f3:<x3,(GUARD3:←ϕ→)>)>)
```

xi represents the internal state of process i, and GUARDi passes exactly those messages destined for Pi; e.g.,

```
DEF GUARD2 == <IF, EQ*[1*1,'P2], [2*1,GUARD2*2], GUARD2*2>
```

The entire system produces a history of the messages passed.

The next section provides some more detailed examples.

4.7 Resource managment in CFFP

CFFP needs no additional constructs for resource managment; instead, some processes serve as supervisors for specific resources. These supervisory processes are similar to the dataflow resource managers of Arvind, Gostelow, and Plouffe [2]. Any external process wishing to access a resource must send a message to the appropriate supervisor which then services the request in some manner. A disk controller and a file manager demonstrate CFFP resource supervisors.

4.7.1 Disk controller

A process retrieves a physical record from a disk by broadcasting a message of the form

<DISK,<ACQUIRE,pname,cylinder,data>> .

DISK indicates that the message is destined for the disk controller[27], ACQUIRE indicates that this is a request for access to a specific location, pname is the name of the requesting process, and cylinder and data define the access. Data contains track and sector specifications, read/write indicators, allocated buffers, and any other required information. The disk controller broadcasts a reply of the form <pname,response> at some later time. This is interprocess communication with the disk controller as one of the communicating processes.

The disk controller executes access requests according to the SCAN algorithm [15]. Conceptually, the arm sweeps in and out across the cylinders, and the controller honors requests as the arm arrives at the cylinder to which a request refers. Figure 21 illustrates a structural algorithm for scan disk control.

- GATE passes only those messages destined for the controller.

- UNITE combines the stream of input messages with the output of the access routine.

- ACCESS is an otherwise unspecified, device specific, physical access procedure.

[27]The controller's gate intercepts all messages, and passes only those directed to the controller.

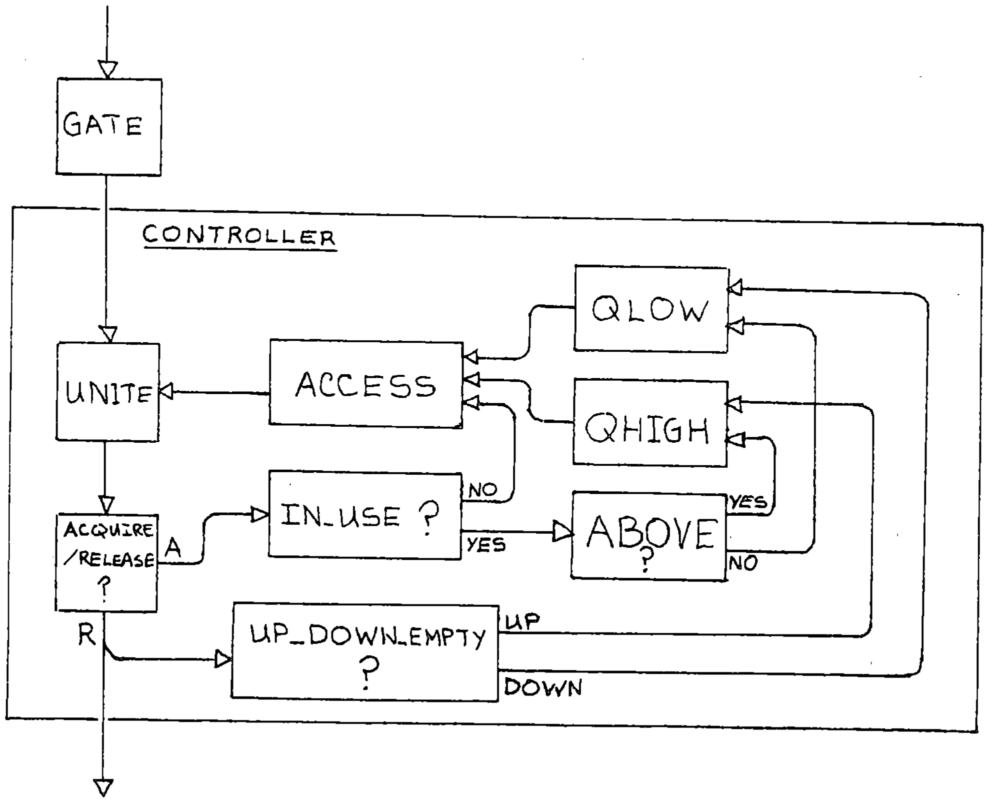


Figure 21. Controller structure

- QHIGH and QLOW are queueing mechanisms for requests to a busy controller. When the current access completes, one of them will be triggered to send an enqueued request to ACCESS.

- The remaining operations route messages and triggers to the appropriate activity based on the type of message (ACQUIRE/RELEASE), the current state of disk activity (IN_USE), the relative positions of the new request and the arm (ABOVE), and the current direction of travel

(UP_DOWN_EMPTY).

These mechanisms allow at most one execution of ACCESS at any time.

```
DEF CONTROLLER == <IF, EQ*[1*1*6,'ACQUIRE], REQUESTS_IN,
                  <IF, EQ*[1*1*6,'RELEASE], RESPONSE_OUT,
                       CONTROLLER*[1,2,3,4,5,2*6]>>

DEF REQUESTS_IN == <IF, IN_USE,
                        <IF, ABOVE, QHIGH, QLOW>,
                        GRANT>

DEF IN_USE == NOT*NULL*3
DEF ABOVE
  == OR*[LT*[2,3*1*6],AND*[EQ*[2,3*1*6],EQ*[1,'DOWN]]]
DEF QHIGH == CONTROLLER*[1,2,3,UPENQ*[TL*1*6,4],5,2*6]
DEF QLOW == CONTROLLER*[1,2,3,4,DOWNENQ*[TL*1*6,5],2*6]
DEF GRANT == CONTROLLER
          *[1,3*1*6,2*1*6,4,5,
              UNITE*[ACCESS*TL*TL*1*6,2*6]]
DEF UPENQ == <IF, NULL*2, APNDL,
             <IF, LT*[2*1,2*1*2], APNDL,
                  APNDL*[1*2,UPENQ*[1,TL*2]]>>
DEF DOWNENQ == <IF, NULL*2, APNDL,
               <IF, GT*[2*1,2*1*2], APNDL,
                    APNDL*[1*2,DOWNENQ*[1,TL*2]]>>
DEF UNITE == <IF, EQ*[1,'1], 2,
                  [1*2*2,UNITE*[1*2,2*2*2]]>*[NEXT,ID]

DEF RESPONSE_OUT == [[3,2*1*6],UP-DOWN-EMPTY]

DEF UP-DOWN-EMPTY == <IF, EQ*[1,'UP],
                          <IF, NULL*4,
                            <IF, NULL*5, RESET, GO_DOWN>,
                            GO_UP>,
                          <IF, NULL*5,
                            <IF, NULL*4, RESET, GO_UP>,
                            GO_DOWN>>

DEF RESET == CONTROLLER*[1,2,'φ,'φ,'φ,2*6]
DEF GO_UP == CONTROLLER
   *['UP,2*1*4,1*1*4,TL*4,5,UNITE*[ACCESS*TL*1*4,2*6]]
DEF GO_DOWN == CONTROLLER
   *['DOWN,2*1*5,1*1*5,4,TL*5,UNITE*[ACCESS*TL*1*5,2*6]]
```

Figure 22. Disk controller

Figure 22 defines the operation of CONTROLLER as a set of CFFP functions. CONTROLLER acts on an argument of the form

```
<direction, position, current-process,
      above-queue, below-queue, signals>  ,
```

and accepts two kinds of signal: external requests to access the disk (ACQUIRE), and internal completion indicators from the current access (RELEASE). If the disk is free, requests are satisfied immediately (GRANT), but if it is in use, the requests are appended to the appropriate waiting queue (QHIGH or QLOW). On a completion, the response is passed out to the appropriate process, and the next waiting request, if any, is serviced (UP-DOWN-EMPTY). If, pl,...,pn represent processes wishing to access the disk, and GATE is defined as

```
DEF GATE == <IF, EQ*[1,1*1*2],
                 [2*1*2,GATE*[1,2*2]],
                 GATE*[1,2*2]>   ,
```

then

```
(⊠|<(CONTROLLER:<UP,1,ϕ,ϕ,ϕ,(GATE:<DISK,<ϕ>>)>),
     pl,...,pn>)
```

instantiates a complete system using our disk controller.

Cycles appear in this system soley as part of the mechanism for interprocess communication. The dapp produces the stream of all messages, and the explicit instance of GATE passes those messages directed to the disk controller.

Instances of GATE within the pi pass those messages directed to each pi. This pattern of communication derives from the interprocess communication mechanism of section 4.6 and generalizes to a hierarchy of communication (i.e., nested dapps of the form (⊠| ...) passing messages to corresponding holes within GATE-applications). Rewriting GATE as

```
DEF GATE == <IF, EQ*['ALL,1*1*2],
                 [2*1*2,GATE*[1,2*2]],
            <IF, EQ*[1,1*1*2],
                 [2*1*2,GATE*[1,2*2]],
                 GATE*[1,2*2]>>
```

adds a form of broadcast message (a message directed to ALL) to the system.

4.7.2 File manager

A file manager should solve the readers/writers problem [12, 23]. This problem envisions an arbitrary combination of readers and writers attempting concurrent access to a common file. Any number of readers may have simultaneous access, but a writer must exclude all other processes when it is active. In addition, the solution must guarantee eventual access to all readers and all writers.

Figure 23 defines a file manager which satisfies the readers/writers requirements. The state of the scheduling algorithm is represented by the number of active readers (ra=item#4), an indicator for active writing (wa=5), a queue

```
DEF MANAGER
  == <IF, EQ*[ENTRY_TYPE,'READ],
          <IF, OK_TO_READ, NEW_READ, HOLD_READ>,
     <IF, EQ*[ENTRY_TYPE,'WRITE],
          <IF, OK_TO_WRITE, NEW_WRITE, HOLD_WRITE>,
     <IF, EQ*[ENTRY_TYPE,'REXIT],
          <IF, CAN_WRITE, OLD_WRITE, READ_DONE>,
     <IF, EQ*[ENTRY_TYPE,'WEXIT],
          <IF, READERS_WAITING, READ_FROM_Q,
          <IF, WRITERS_WAITING,
               WRITE_FROM_Q, WRITE_DONE>>,
      CONTINUE>>>>

DEF ENTRY_TYPE == 1*1*6
DEF OK_TO_READ == AND*[NOT*5,NULL*3]
DEF OK_TO_WRITE == AND*[EQ*[4,'0],NOT*5]
DEF CAN_WRITE == AND*[EQ*[4,'1],NOT*NULL*3]
DEF READERS_WAITING == NOT*NULL*2
DEF WRITERS_WAITING == NOT*NULL*3
DEF CONTINUE == MANAGER*[1,2,3,4,5,2*6]

DEF NEW_READ == [['CREATE,XR*[1,2*1*6]],
                 MANAGER*[1,2,3,ADD*[4,'1],5,2*6]]
DEF HOLD_READ == MANAGER*[1,APNDR*[2,2*1*6],3,4,5,2*6]

DEF NEW_WRITE == [['CREATE,XW*[1,2*1*6]],
                  MANAGER*[1,2,3,4,'T,2*6]]
DEF HOLD_WRITE == MANAGER*[1,2,APNDR*[3,2*1*6],4,5,2*6]

DEF OLD_WRITE ==[['CREATE,XW*[1,1*3]],
                 MANAGER*[1,2,TL*3,'0,'T,2*6]]
DEF READ_DONE == MANAGER*[1,2,3,SUB*[4,'1],5,2*6]

DEF READ_FROM_Q
  == <IF, NULL*2, MANAGER*[2*1*6,'φ,3,4,'F,2*6],
          [['CREATE,XR*[2*1*6,1*2]],
           READ_FROM_Q*[1,TL*2,3,ADD*[4,'1],5,6]]>
DEF WRITE_FROM_Q == [['CREATE,XW*[2*1*6,1*3]],
                     MANAGER*[2*1*6,2,TL*3,'0,'T,2*6]]
DEF WRITE_DONE == MANAGER*[2*1*6,2,3,4,'F,2*6]
```

Figure 23. File manager

of waiting readers (rq=2), and a queue of waiting writers (wq=3). In addition, MANAGER maintains a copy of the file (file=1) and an entry point for messages (entry=6). Thus, MANAGER continuously acts on an argument of the form

<file, rq, wq, ra, wa, entry>

As in the disk controller, the entry point is the result of gating a global hole, (GATE:<FILE,$\Leftarrow\phi,\Rightarrow$>).

MANAGER accepts four message classes, services them, and changes its state as appropriate.

1. <READ,<source,data>>. A request by an independent process to read from the file. Source is the origin of the request, and data gives the details of the request. If no writers are active or waiting, OK_TO_READ, the request is immediately honored, NEW_READ; otherwise, the request is appended to the end of the waiting readers queue, HOLD_READ. MANAGER honors read requests by spawning a subsidiary process, (XR:<file,<source,data>>), and incrementing the count of active readers. XR is an unspecified operation which reads the file and evaluates to

   <<FILE,<REXIT>>,<<source,read-result>,ϕ>>

   This result is a message to the file manager indicating that the read is done, followed by a message to the original process with the results of the read, and a proper termination (ϕ).

2. <WRITE,<source,data>> is a request to write to the file. The request is honored if the file is

inactive and enqueued otherwise. MANAGER honors write requests with the subsidiary process (XW:<file,<source,data>>) which evaluates to

```
        <<FILE,<WEXIT,new-file>>,
                  <<source,write-result,ϕ>>>  .
```

New-file is the file as modified by the write. When writing, wa is set to "T".

3. <REXIT> indicates the completion of a read operation. If no other readers are active, MANAGER honors the request of the first waiting writer.

4. <WEXIT,new-file> denotes the end of writing; new-file replaces the internal copy of the file. MANAGER grants access to <u>all</u> waiting readers, or if there are no waiting readers, it instantiates the first waiting writer.

⊠ cannot supervise the entire system since MANAGER is constantly creating subsidiary processes; instead, we define a new SYSTEM operation.

```
DEF SYSTEM == <IF, NULL*1, 'ϕ,
              <IF, EQ*[1*1*APPLY, 'CREATE],
                   SYSTEM*APNDL*[2*1*APPLY,NONULLS*REST],
                   [1*APPLY,SYSTEM*NONULLS*REST]>>
                *[NEXT,ID]
```

SYSTEM acts just like ⊠ except that "messages" with destination CREATE are held as internal processes. If p1,...,pn are processes which access a common file through MANAGER, the complete system begins as

```
(SYSTEM|<(MANAGER:<φ,φ,φ,0,F,(GATE:<FILE,<φ>>)>),
         p1,...,pn>)
```

The above file manager (based on [23]) prevents starvation of both readers and writers. As with dataflow resource managers, this supervisory process models two other solutions with only minor modifications.

1. Readers have priority provided no writer is active [12]. To implement this constraint, only OK_TO_READ needs modification.

   ```
   DEF OK_TO_READ == NOT*5
   ```

   The resulting manager may indefinitely postpone writers.

2. Writers have priority if no reader is active [12]. The solution now requires exchanging the positions in MANAGER of READERS_WAITING with WRITERS_WAITING, and of READ_FROM_Q with WRITE_FROM_Q. This solution allows readers to starve.

The disk controller and file manager indicate a way of writing modules for CFFP-based systems, and the versatility of the file manager suggests that such systems are relatively easy to modify.

CHAPTER 5

Conclusions and future research directions

CFFP is an applicative computing language derived from FFP through three major alterations.

1. Applications may execute as soon as they are minimally defined. For example, (1:<(ADD:<3,5>),7) may execute to produce (ADD:<3,5>). Such an application may persist over possibly infinite streams of inputs by performing a calculation on the heads of the streams, and then reinstantiating itself on their tails. This persistence is precisely the behavior we require from history-sensitive processes. In addition, CFFP is more asynchronous than FFP since dependent applications may often act in parallel.

2. An indeterminate operation, $\underline{r}$NEXT, joins the list of primitive functions. This operation often represents the random order of message production and reception between cooperating processes, and our descriptions of systems of communicating processes rely on this new primitive and cycles (the third alteration). The assertion that all

copies of the same expression have the same meaning and partial meanings allows the consistent use of NEXT in our history-sensitive system. In the history-sensitive system, several copies of the same application may exist, and should have the same meaning; to retain this quality, we limit the indeterminacy of NEXT by insisting that all copies of the same expression continue to produce the same results. This form of indeterminacy only represents an external view of the operation and not its internal nature.

3. Distinguished applications and holes represent explicit cycles. Since recursion produces new instances and not copies, we must rely on this structure to provide consistent, controlled cyclic communication.

With these alterations, an application can serve as a history-sensitive process such as a resource manager (e.g., file manager) or an entire system in which independent processes act. In this way, CFFP provides a homogeneous representation for simple computations, processes, and systems. This homogeneous representation contrasts sharply with AST systems [6] where applications only represent computations, a state-changing mechanism activates applications and denotes a process, and no complete representation exists for a system made up of two or more communicating processes.

Two modifications would improve CFFP.

1. Some form of cycle-forming operation (similar to the function represented by APPLY) would allow dynamic creation of general subsystems. With any such operation, we must also develop a notion of well-formed cycle-formation to accompany the concept of well-formed dapp.

2. Holes should behave more like applications. Currently, the partial meaning of a hole is the result at the hole when $\underline{p}$ finds the partial meaning of the hole's dapp; i.e.,

   $$\underline{p}\langle h\rangle == \{\underline{k}\langle h\rangle\}\{\underline{p}\{\{\theta\langle h\rangle\}\langle h\rangle\}\} \quad .$$

   Ideally, $\underline{p}\langle h\rangle$ should affect only $\langle h\rangle$ and not the entire context of $\langle h\rangle$ (i.e., $\{\theta\langle h\rangle\}\langle h\rangle$).

In addition, three elaborations would enhance CFFP: 1) a set of formal conditions guaranteeing that a dapp is (or is not) well-formed; 2) an algebra of programs similar to that of Backus for FP programs [6, 7]; 3) an alternate notion of meaning which would include such non-terminating computations as well-formed dapps.

CFFP is a fundamental model of computing, and can serve as the basis for research in at least two directions:

1. It supports new approaches to computer architecture. Either Mago's machine [27, 28] or Frank's machine [17] could serve as a starting point in this direction.

2. A higher-level, user-oriented language could use CFFP as its target. Conventional expressions could denote applications:

```
3+4          (ADD:<3,4>)
2*8+7        (ADD:<(TIMES:<2,8>),7>)
7*(6+13)     (TIMES:<7,(ADD:<6,13>)>)

if 12<17 then 17-12 else 12-17
             (<IF,LT,SUB*[2,1],SUB>:<12,17>)
```

A variable could represent one expression within others.

- An assignment statement could form the association between the variable and the expression.

  ```
  x := 3 + 8
  ```

  This assignment would not represent storage of a value in a location; instead, it would specify that x represents 3+8 in other expressions.

  ```
  x * 5 == (3 + 8) * 5
  ```

  Only one assignment to a particular variable should occur in any program. Other pseudo-algorithmic, expression-oriented languages (Id [3] and Lucid [4]) have successfully used this single-assignment system.

- With such variables, we may borrow some notation from Landin [26] to specify complex applications.

```
u*(u+1)-v*(v+1)
  where u := 2*p+q;
        v := p-2*q;
        p := 33;
        q := 21

(SUB*[TIMES*[1,ADD*[1,'1]],
    TIMES*[2,ADD*[2,'1]]]
  *[ADD*[TIMES*['2,1],2],
    SUB*[1,TIMES*['2,2]]]:<33,21>)
```

A higher-level language should also contain a function-defining facility

- A special assignment statement could form a definition.

```
fun absdif(a,b) := if a<b then b-a else a-b

    DEF ABSDIF == <IF,LT,SUB*[2,1],SUB>
```

- An expression could directly use such a definition,

```
subr(u)-subr(v)
where fun subr(x) := x*(x+1);
      u := 2*p+q;
      v := p-2*q;
      p := 33;
      q := 21
```

and/or the definition could become part of the evaluation mechanism.

Expressions, pseudo-variables, and definitions are only a starting point toward building a higher-level language, but they do suggest an approach that might yield such a language.

Backus [7] argues that function level programs are easier to understand and analyze than are object level programs. A function level program is a primitive function or a combination over functions and function-valued variables, and an object level program includes data-valued objects. Any higher-level derivative of CFFP should be a function level language or should have a simple direct translation to the function level. Backus' lift mapping [7] describes an object level to function level transformation.

Bibliography

1. Arvind, and Gostelow, K.P. Microelectronics and computer science. Proc. Second IEEE (G-PHP)/ISHM University/Industry/Government Microelectronics Symp., U. of New Mexico, Albuquerque, January, 1977.

2. Arvind, Gostelow, K.P., and Plouffe, W. Indeterminacy, Monitors, and Dataflow. Proc. Sixth ACM Symp. on Operating Systems Principles, Purdue U., November, 1977, pp. 159-169.

3. Arvind, Gostelow, K.P., and Plouffe, W. An asynchronous programming language and computing machine. Tech. Rep. No. 114, Dept. Info. and Comptr. Sci., U. of California, Irvine, May, 1978.

4. Ashcroft, E.A., and Wadge, W.W. Lucid--A formal system for writing and proving programs. SIAM J. Comput. 5, 3 (September 1976), 336-354.

5. Backus, J. Programming language semantics and closed applicative languages. Conf. Record ACM Symp. on Principles of Programming Languages, Boston, October, 1973, pp. 71-86.

6. 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.

7. Backus, J. Function level programs as mathematical objects. Proc. 1981 conf. on Functional Programming Languages and Computer Architecture, Portsmouth, NH, October, 1981, pp. 1-10.

8. Berkling, K.J. A symetric complement to the lambda calculus. Interner Bericht ISF-76-7, Gesellschaft fur Mathematik und Datenverarbeitung MBH, September, 1976.

9. Berkling, K.J. Reduction languages for reduction machines. Interner Bericht ISF-76-8, Gesellschaft fur Mathematik und Datenverarbeitung MBH, September, 1976.

10. Burks, A.W., Goldstine, H.H., and von Neumann, J. Preliminary discussion of the logical design of an electronic computing instrument. In Collected Works of John von Neumann, Volume 5, A.H. Taub, Ed., 1963, pp. 34-79. Taken from report to U.S. Army Ordinance Department, 1946

11. Church, A. Annals of Mathematics Studies. Volume 6: The Calculi of Lambda-Conversion. Princeton University Press, Princeton, 1941.

12. Courtois, P.J., Heymans R., and Parnas, D.L. Concurrent control with "readers" and "writers". Comm. ACM 14, 10 (October 1971), 667-668.

13. Curry, H.B., and Feys, R. Combinatory Logic, Volume I. North-Holland, Amsterdam, 1958.

14. Curry, H.B., Hindley, J.R., and Seldin, J.P. Combinatory Logic, Volume II. North-Holland, Amsterdam, 1972.

15. Denning, P.J. Effects of scheduling on file memory operations. Proc. AFIPS 1967 SJCC 30, 1967, pp. 9-21.
16. Dennis, J.B. First version of a data flow procedure language. Tech. Mem. No. 61, Lab. for Comptr. Sci., M.I.T., May, 1973.
17. Frank, G.A. *Virtual memory systems for closed applicative language interpreters*. Ph.D. Th., U. of N. Carolina, Chapel Hill, 1979.
18. Friedman, D.P., and Wise, D.S. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, S. Michaelson and R. Milner, Eds., Edinburgh U. Press, Edinburgh, 1976, pp. 257-284.
19. Friedman, D.P., and Wise, D.S. The impact of applicative programming on multiprocessing. Proc. Intl. Conf. on Parallel Processing, 1976, pp. 263-272.
20. Friedman, D.P., and Wise, D.S. A constructor for applicative multiprogramming. Tech. Rep. No. 80, Comptr. Sci. Dept., Indiana U., January, 1979.
21. Friedman, D.P., and Wise, D.S. Applicative multiprogramming. Tech. Rep. No. 72, Comptr. Sci. Dept., Indiana U., April, 1979.
22. Henderson, P., and Morris, J.H. Jr. A lazy evaluator. Conf. Record Third ACM Symp. on Principles of Programming Languages, Atlanta, January, 1976, pp. 95-103.
23. Hoare, C.A.R. Monitors: An operating system structuring concept. *Comm. ACM 17*, 10 (October 1974), 549-557.
24. Keller, R.M. Semantics of parallel program graphs. Tech. Rep. No. 77-110, Dept. Comptr. Sci., U. of Utah, July, 1977.
25. Keller, R.M., Lindstrom, G., and Patil S. An architecture for a loosely-coupled parallel processor. Tech. Rep. No. 78-105, Dept. Comptr. Sci., U. of Utah, October, 1978.
26. Landin, P.J. The mechanical evaluation of expressions. *Computer J. 6*, 4 (1964), 308-320.
27. Mago, G.A. A network of microprocessors to execute reduction languages, pt. I. *Intl. J. Comptr. and Info. Sci. 8*, 5 (October 1979), 349-385.
28. Mago, G.A. A network of microprocessors to execute reduction languages, pt. II. *Intl. J. Comptr. and Info. Sci. 8*, 6 (December 1979), 435-471.
29. Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, San Francisco, 1974.
30. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, pt. I. *Comm. ACM 3*, 4 (April 1960), 184-195.

31. Schonfinkel, M. Uber di Bausteine der mathematischen Logik. Mathematical Annals 92 (1924), 305-316. English translation in From Frege to Godel; A Source-Book in Mathematical Logic, Harvard U. Press. 1967

32. Shaw, A.C. The Logical Design of Operating Systems. Prentice-Hall, Englewood Cliffs, NJ, 1974.

33. Stoy, J.E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, 1977.

34. Tonge, F.M., and Cowan, R.M. Structured process description. Tech. Rep. No. 130, Dept. Info. and Comptr. Sci., U. of California, Irvine, May, 1980.

35. Turner, D.A. A new implementation technique for applicative languages. Software--Practice and Experience 9 (1979), 31-49.

36. Weng, K.-S. Stream-oriented computation in recursive data flow schemas. Master Th., M.I.T., October 1975.

APPENDIX I

Primitive functions

The following list of primitive operations includes all those of Backus [5, 6] and this dissertation. In general, the representation is given in terms of the interacting semantics. These representations become appropriate to standard FFP when internal evaluations are disallowed, and to CFFP when evaluations for component dapps and holes are added. EQ is not valid under the interacting semantics or CFFP, and its representation is under standard FFP. F, IF, T and WHILE significantly change from standard to more-defined FFP and we also provide the standard FFP representation for each of them. $\underline{N}$ is the set of numbers.

```
{rAA}x == x=<<z,f>,<y1,...,yn>> => <(f:y1),...,(f:yn)>;
     x=<(f:z),y> => {rAA}<p(f:z),y>;
     x=<<z,f>,(g:u)> => {rAA}<<z,f>,p(g:u)>;
     x=(f:z) => {rAA}{px}

{rADD}x == x=<n1,n2>, n1,n2∈N => n1+n2;
     x=<(f:z),n> => {rADD}<p(f:z),n>;
     x=<n,(f:z)> => {rADD}<n,p(f:z)>;
     x=(f:z) => {rADD}{px}

{rAND}x == x=<T,T> => T;
     x=<y,z>, y,z∈{T,F} => F;
     x=<(f:z),y> => {rAND}<p(f:z),y>;
     x=<y,(f:z)> => {rAND}<y,p(f:z)>;
     x=(f:z) => {rAND}{px}
```

```
{rAPNDL}x == x=<y,ϕ> => <y>;
     x=<y,<x1,...,xn>> => <y,x1,...,xn>;
     x=<y,(f:z)> => {rAPNDL}<y,p(f:z)>;
     x=(f:z) => {rAPNDL}{px}


{rAPNDR}x == x=<ϕ,y> => <y>;
     x=<<x1,...,xn>,y> => <x1,...,xn,y>;
     x=<(f:z),y> => {rAPNDR}<p(f:z),y>;
     x=(f:z) => {rAPNDR}{px}


{rAPPLY}x == x=<f,y> => (f:y);
     x=(g:z) => {rAPPLY}{px}


{rAL}x == x=<<z,f>,<y1,...,yn>> => <(f:y1),y2,...,yn>;
     x=<(f:z),y> => {rAL}<p(f:z),y>;
     x=<<z,f>,(g:u)> => {rAL}<<z,f>,p(g:u)>;
     x=(f:z) => {rAL}{px}


{rAR}x == x=<<z,f>,<y1,...,yn,u>> => <y1,...,yn,(f:u)>;
     x=<(f:z),y> => {rAR}<p(f:z),y>;
     x=<<z,f>,(g:u)> => {rAR}<<z,f>,p(g:u)>;
     x=(f:z) => {rAR}{px}


{rATOM}x == x∊A => T;
     x=<x1,...,xn> => F;
     x=(f:z) => {rATOM}{px}


{rBU}x == x=<<z,f,u>,v> => (f:<u,v>);
     x=<(f:z),y> => {rBU}<p(f:z),y>;
     x=(f:z) => {rBU}{px}


{rCN}x == x=<T,<u,v>> => u;
     x=<F,<u,v>> => v;
     x=<(f:z),y> => {rCN}<p(f:z),y>;
     x=<b,(f:z)> => {rCN}<b,p(f:z)>;
     x=(f:z) => {rCN}{px}


{rCOMP}x == <<z,f1,...,fn>,y> => (f1:(...}(fn:y)...));
     x=<(f:z),y> => {rCOMP}<p(f:z),y>;
     x=(f:z) => {rCOMP}{px}
```

```
{rCONS}x == <<z,f1,...,fn>,y> => <(f1:y),...,(fn:y)>;
     x=<(f:z),y> => {rCONS}<p(f:z),y>;
     x=(f:z) => {rCONS}{px}


{rCONST} == <<z,c>,y> => c;
     x=<(f:z),y> => {rCONST}<p(f:z),y>;
     x=(f:z) => {rCONST}{px}


{rDBL}x == <x,x>


{rDISTL}x == x=<y,φ> => φ;
     x=<y,<x11,...,xn>> => <<y,x1>,...,<y,xn>>;
     x=<y,(f:z)> => {rDISTL}<y,p(f:z)>;
     x=(f:z) => {rDISTL}{px}


{rDISTR}x == x=<φ,y> => φ;
     x=<<x1,...,xn>,y> => <<x1,y>,...,<xn,y>>;
     x=<(f:z),y> => {rDISTR}<p(f:z),y>;
     x=(f:z) => {rDISTR}{px}


{rDIV}x == x=<n1,n2>, n1,n2∈N => n1/n2;
     x=<(f:z),n> => {rDIV}<p(f:z),n>;
     x=<n,(f:z)> => {rDIV}<n,p(f:z)>;
     x=(f:z) => {rDIV}{px}


{rEQ}x == <y,y>, y≠⊥ => T;    standard FFP only
     x=<u,v>, u,v≠⊥ => F


{rEQAT}x == x=<y,y>, y∈A => T;
     x=<y,z>, y,z∈A => F;
     x=<(f:z),y> => {rEQAT}<p(f:z),y>;
     x=<y,(f:z)> => {rEQAT}<y,p(f:z)>;
     x=(f:z) => {rEQAT}{px}


{rF}x == x=<y,z> => z;
     x=(f:z) => {rF}

     standard FFP:
     {rF}x == x=<y,<f,z>> => (f:z)
```

```
{rFST1}x == x=<<u,v>,<y,z>> => <(v:y),z>;
     x=<(f:z),y> => {rFST1}<p(f:z),y>;
     x=<y,(f:z)> => {rFST1}<y,p(f:z)>;
     x=(f:z) => {rFST1}{px}


{rID}x == x


{rIF}x == x=<<z,p,f,g>,y> => ((p:y)}<(f:y),(g:y)>);
     x=<(f:z),y> => {rIF}<p(f:z),y>;
     x=(f:z) => {rIF}{px}

     standard FFP:
     {rIF}x == x=<<z,p,f,g>,y> => ((p:y)}<<f,y>,<g:y>>)


{rINSERT}x == x=<<z,f>,<y>> => y;
     x=<<z,f>,<y1,...,yn>> => (f:(<INSERT,f>:<y2,...,yn>));
     x=<(f:z),y> => {rINSERT}<p(f:z),y>;
     x=<<z,f>,(g:u)> => {rINSERT}<<z,f>,p(g:u)>;
     x=(f:z) => {rINSERT}{px}


{rLENGTH}x == x=ϕ => 0;
     x=<x1,...,xn> => n;
     x=(f:z) => {rLENGTH}{px}


{rNEXT}x == x=ϕ => ϕ;
     x=<x1,...,xn> => {{∃xi}pxi=<y1,y2>} => i;
                        {∀xi}pxi=ϕ} => ϕ; ⊥};
     x=(f:z) => {rNEXT}{px}


{rNOT}x == x=T => F; x=F => T; x=(f:z) => {rNOT}{px}


{rNULL}x == x=ϕ => T;
     x=<x1,...,xn> => F;
     x∈A => F;
     x=(f:z) => {rNULL}{px}


{rOR}x == x=<F,F> => F;
     x=<y,z>, y,z∈{T,F} => T;
     x=<(f:z),y> => {rOR}<p(f:z),y>;
     x=<y,(f:z)> => {rOR}<y,p(f:z)>;
     x=(f:z) => {rOR}{px}
```

```
{rREVERSE}x == x=ϕ => ϕ;
     x=<x1,...,xn> => <xn,...,x1>;
     x=(f:z) => {rREVERSE}{px}


{rROTL}x == x=ϕ => ϕ;
     x=<y> => <y>;
     x=<x1,...,xn> => <x2,...,xn,x1>;
     x=(f:z) => {rROTL}{px}


{rROTR}x == x=ϕ => ϕ;
     x=<y> => <y>;
     x=<x1,...,xn,y> => <y,x1,...,xn>;
     x=(f:z) => {rROTR}{px}


{rSEPL}x == x=<y> => <y,ϕ>;
     x=<x1,...,xn> => <x1,<x2,...,xn>>;
     x=(f:z) => {rSEPL}{px}


{rSEPR}x == x=<y> => <ϕ,y>;
     x=<x1,...,xn,y> => <<x1,...,xn>,y>;
     x=(f:z) => {rSEPR}{px}


{rSUB}x == x=<n1,n2>, n1,n2∈N => n1-n2;
     x=<(f:z),n> => {rSUB}<p(f:z),n>;
     x=<n,(f:z)> => {rSUB}<n,p(f:z)>;
     x=(f:z) => {rSUB}{px}


{rT}x == x=<y,z> => y;
     x=(f:z) => {rT}{px}

     standard FFP:
     {rT}x == <<f,y>,z> => (f:y)


{rTIMES}x == x=<n1,n2>, n1,n2∈N => n1*n2;
     x=<(f:z),n> => {rTIMES}<p(f:z),n>;
     x=<n,(f:z)> => {rTIMES}<n,p(f:z)>;
     x=(f:z) => {rTIMES}{px}


{rTL}x == x=<y> => ϕ;
     x=<x1,...,xn> => <x2,...,xn>;
     x=(f:z) => {rTL}{px}
```

```
{rTLR}x == x=<y> => φ;
     x=<x1,...,xn,y> => <x1,...,xn>;
     x=(f:z) => {rTLR}{px}


{rTRANS}x == x=<φ,...,φ> => φ;
     x=<x1,...,xn>
       => {∀i:xi=<yi1,...,yim>
             => <z1,...,zm>   zj=<y1j,...,ynj>;
           ǂi:xi=(f:z) => {rTRANS}<px1,...,pxn>; ⊥};
     x=(f:z) => {rTRANS}{px}


{rWHILE}x == x=<<z,p,f>,y>
               => ((p:y)}<(<WHILE,p,f>}(f:y)),y>);
     x=<(f:z),y> => {rWHILE}<p(f:z),y>;
     x=(f:z) => {rWHILE}{px}

     standard FFP:
     {rWHILE}x
       == x=<<z,p,f>,y>
           => ((p:y)}<<<COMP,<WHILE,p,f>,f>,y>,<ID,y>>)


{rs}x == x=<x1,...,xs,...,xn> => xs;
     x=(f:z) => {rs}{px}


{rsR}x == x=<xn,...,xs,...,x1> => xs;
     x=(f:z) => {rsR}{px}
```

APPENDIX II

Formal properties

Three of Backus six formal properties [5] are observations; these are the "anti-quote" property, non-extensionality, and single-type functions. We prove the remaining three properties for CFFP as follows.

1. <u>idempotency</u>. $\underline{m}\{\underline{m}x\} = \underline{m}x$.
   We enumerate the possible forms of x.

   case 1: $x \in \underline{A} \Rightarrow \underline{m}x = x \Rightarrow \underline{m}\{\underline{m}x\} = \underline{m}x$

   case 2: $x = \underline{\bot} \Rightarrow \underline{m}x = x \Rightarrow \underline{m}\{\underline{m}x\} = \underline{m}x$

   case 3: $x = \underline{w} \Rightarrow \underline{m}x = x \Rightarrow \underline{m}\{\underline{m}x\} = \underline{m}x$

   case 4: $x \in \underline{H} \Rightarrow \underline{m}x = \underline{w} \Rightarrow \underline{m}\{\underline{m}x\} = \underline{mw} = \underline{w} = \underline{m}x$

   case 5: $$\begin{aligned} x &= \langle x1, \ldots, xn\rangle \\ &\Rightarrow \underline{m}x = \langle \underline{m}x1, \ldots, \underline{m}xn\rangle \\ &\Rightarrow \underline{m}\{\underline{m}x\} = \langle \underline{m}\{\underline{m}x1\}, \ldots, \underline{m}\{\underline{m}xn\}\rangle \\ &\qquad = \langle \underline{m}x1, \ldots, \underline{m}xn\rangle \\ &\qquad\qquad \text{by structural induction} \\ &\qquad = \underline{m}x \end{aligned}$$

   case 6: $x = (f{:}y) \Rightarrow \underline{m}x = \underline{m}\{\underline{p}x\}$

   $$\begin{aligned} &\underline{p}x \in \underline{A} \cup \{\langle x1,\ldots,xn\rangle\} \cup \{\underline{\bot}\} \cup \{\underline{w}\} \\ &\quad \Rightarrow \underline{m}\{\underline{m}x\} = \underline{m}\{\underline{m}\{\underline{p}x\}\} = \underline{m}\{\underline{p}x\} = \underline{m}x \end{aligned}$$

   case 7: $x = (f|y) \Rightarrow \underline{m}x = \underline{m}\{\underline{p}x\}$

   $$\begin{aligned} &\underline{p}x \in \{\phi\} \cup \{\langle x1,\ldots,xn\rangle\} \cup \{\underline{w}\} \\ &\quad \Rightarrow \underline{m}\{\underline{m}x\} = \underline{m}\{\underline{m}\{\underline{p}x\}\} = \underline{m}\{\underline{p}x\} = \underline{m}x \end{aligned}$$

2. <u>the Church-Rosser property</u>.

   a. $x \text{-->>} y;\ x \text{-->>} z;\ y, z \in \underline{C} \Rightarrow y = z$

   b. $x \text{-->>} y \Rightarrow y \text{-->>} \underline{m}x$

We begin by formalizing the notion of transition (reduction), and observe that any transition is the substitution of the execution of a component application or dapp for that application or dapp. The set $\underline{X}$ denotes executable applications and dapps.

Definition 1.

```
$\underline{X}$ == {(f:x)|f∈$\underline{A}$,$\underline{d}$f=#,f∈$\underline{P}$i,x∈$\underline{A}$i}
              U {(f:x)|f∈$\underline{A}$,$\underline{d}$f=g≠#} U {(⊥:x)}
              U {(f:x)|f=<f1,...,fn>}
              U {(f|x)∈$\underline{Z}$|(f:x)∈$\underline{X}$}
```

where the sets $\underline{P}$i and $\underline{A}$i are those of figure 6. [x/y]z denotes the substitution of x for a particular y in z, and $\underline{x}$:$\underline{X}$-->$\underline{E}$ is the execution function.

Definition 2.

```
$\underline{x}$x == x=(f:y)
         => {f∈$\underline{A}$ => {$\underline{d}$f=# => {$\underline{r}$f}y;
                          $\underline{d}$f=g => (g:y)};
              f=<f1,...,fn> => (f1:<f,y>);
              f=⊥ => ⊥};
        x=(f|y)∈$\underline{Z}$
         => {f∈$\underline{A}$ => {$\underline{d}$f=# => { {$\underline{r}$f}y=(g:z)
                                      => (g|z);
                                      $\underline{y}${{$\underline{r}$f}y} };
                          $\underline{d}$f=g => (g|y) };
              f=<f1,...,fn> => (f1|<f,y>)}
```

We next establish a lemma which asserts that execution does not change partial meaning.

Lemma 1. $\underline{p}\{\underline{x}x\} = \underline{p}x$

There are nine valid cases

case 1. $x=(f:y),\ f\in\underline{A},\ \underline{d}f=\#$
$\underline{p}x = \underline{p}\{\{\underline{r}f\}y\} = \underline{p}\{\underline{x}x\}$

case 2. $x=(f:y),\ f\in\underline{A},\ \underline{d}f=g\neq\#$
$\underline{p}x = \underline{p}(g:y) = \underline{p}\{\underline{x}x\}$

case 3. $x=(f:y),\ f=\langle f1, \ldots ,fn\rangle$
$\underline{p}x = \underline{p}(f1:\langle f,y\rangle) = \underline{p}\{\underline{x}x\}$

case 4. $x=(\bot:y)$
$\underline{p}x = \bot = \underline{p}\{\underline{x}x\}$

case 5. $x=(f|y),\ f\in\underline{A},\ \underline{d}f=\#,\ \{\underline{r}f\}y=(g:z)$
$\underline{p}x = \underline{y}\{\underline{p}(f:y)\} = \underline{y}\{\underline{p}(g:z)\} = \underline{p}(g|z)$
$= \underline{p}\{\underline{x}x\}$

case 6. $x=(f|y),\ f\in\underline{A},\ \underline{d}f=\#,\ \{\underline{r}f\}y=\phi$
$\underline{p}x = \underline{y}\{\underline{p}(f:y)\} = \underline{y}\phi = \phi = \underline{p}\phi$
$= \underline{p}\{\underline{x}x\}$

case 7. $x=(f|y),\ f\in\underline{A},\ \underline{d}f=\#,\ \{\underline{r}f\}y=\langle g,z\rangle$
$\underline{p}x = \underline{y}\{\underline{p}(f:y)\} = \underline{y}\{\underline{p}\langle z,g\rangle\} = \underline{y}\langle z,g\rangle$
$= \langle z',g'\rangle = \underline{p}\langle z',g'\rangle = \underline{p}\{\underline{y}\langle z,g\rangle\}$
$= \underline{p}\{\underline{x}x\}$

case 8. $x=(f|y),\ f\in\underline{A},\ \underline{d}f=g\neq\#$
$\underline{p}x = \underline{y}\{\underline{p}(f:y)\} = \underline{y}\{\underline{p}(g:y)\}$
$= \underline{p}(g|y) = \underline{p}\{\underline{x}x\}$

case 9. $x=(f|y),\ f=\langle f1,\ldots,fn\rangle$
$\underline{p}x = \underline{y}\{\underline{p}(f1:\langle f,y\rangle)\}$
$= \underline{p}(f1|\langle f,y\rangle) = \underline{p}\{\underline{x}x\}$

As a corollary we show that whenever x has no partial meaning, neither does its execution (M' is the set of expressions with partial meanings).

Corollary 1. $x \notin M' \Rightarrow \underline{x}x \notin M'$
proof: $\underline{x}x \in M' \Rightarrow \underline{p}x = \underline{p}\{\underline{x}x\} \Rightarrow x \in M'$

We now need an ordering on expressions, and choose the partial meaning as the basis for this ordering. Any expression is less defined than or equal to ($\leq$) its partial meaning, and any expression with components $\leq$ all those of a second expression is also $\leq$ that second expression; formally,

$$\begin{aligned}
&\forall x \in \{\underline{w}\} \cup \underline{E}:\ \underline{w \leq x} \\
&\forall x \in \underline{E}:\ \underline{\bot \leq x},\ x \underline{\leq p}x \\
&\forall k \in \underline{K}:\ xj \leq xj' \Rightarrow k[x1,\ldots,xj,\ldots,xn] \\
&\qquad\qquad\qquad\qquad \leq k[x1,\ldots,xj',\ldots,xn]
\end{aligned}$$

Under this ordering, $\underline{m}$, $\underline{p}$, and all primitive functions are continuous, and this leads directly to lemma 2.

Lemma 2. $x \in M,\ y \in M' \Rightarrow \underline{m}x = \underline{m}\{[\underline{p}y/y]x\}$.

$$\begin{aligned}
&y \leq \underline{p}y \Rightarrow \underline{m}x \leq \underline{m}\{[\underline{p}y/y]x\} \\
&\underline{m}x \neq \underline{m}\{[\underline{p}y/y]x\} \Rightarrow \exists c \in \underline{C}:\ c \neq \underline{p}c \\
&\qquad \text{contradiction}
\end{aligned}$$

Based on lemmma 2, we prove lemma 3.

Lemma 3. $x \dashrightarrow x' \Rightarrow \underline{m}x = \underline{m}x'$

$$\begin{aligned}
x' &== [\underline{x}y/y]x \\
\underline{m}x' &= \underline{m}\{[\underline{x}y/y]x\} \\
&= \underline{m}\{[\underline{p}\{\underline{x}y\}/\underline{x}y]\{[\underline{x}y/y]x\}\} && \text{(lemma 2)} \\
&= \underline{m}\{[\underline{p}\{\underline{x}y\}/y]x\} \\
&= \underline{m}\{[\underline{p}y/y]x\} && \text{(lemma 1)} \\
&= \underline{m}x && \text{(lemma 2)}
\end{aligned}$$

Lemma 3 immediately results in the Church-Rosser property since

$$x \text{-->>} y,\ x \text{-->>} z \ \Rightarrow\ \underline{m}y = \underline{m}z = \underline{m}x$$

by induction on the number of transitions, and

$$y = \underline{m}y,\ z = \underline{m}z \ \Rightarrow\ y=z=\underline{m}x$$

Part b follows since $y \text{-->>} \underline{m}y$ by replacing each execution step in $\underline{m}y$ by a transition.

3. *the reduction property*. $\underline{m}\{[\underline{m}y/y]x\} = \underline{m}x$.

This is essentially a corollary to Church-Rosser. By providing a transition for each execution in $\underline{m}y$ we have

$$x \text{-->>} [\underline{m}y/y]x$$

and

$$\underline{m}x = \underline{m}\{[\underline{m}y/y]x\} \text{ by lemma 3.}$$