A STUDY OF THE FUNCTIONAL PROGRAMMING LANGUAGE FP

Beverly P. Rollins
B.Mus., Indiana University, 1973
M.Mus., Indiana University, 1977

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science

August, 1983

The thesis "A Study of the Funtional Programming Language FP" by Beverly P. Rollins has been examined and approved by the following Examination Committee:

Richard B. Kieburtz, Thesis Research Advisor
Professor,
Dept. of Computer Science and Engineering

Robert G. Babb
Assistant Professor,
Dept. of Computer Science and Engineering

David Maier
Associate Professor,
Dept. of Computer Science and Engineering

James Hein
Associate Professor,
Dept. of Computer Science, Portland State University

DEDICATION

I dedicate this thesis to my husband Gene, my best friend and teacher.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Richard Kieburtz, for his patience, guidance, and enthusiasm. I would also like to thank Jon Shultis for introducing me to functional programming, and John Givler for always being available to answer my questions.

Thank you, also, fellow students, for sharing ideas, and thanks to the staff at the Oregon Graduate Center for providing a good environment for us to work in. Special thanks go to Carla Rathbun and Bruce Jerrick, who do a wonderful job of managing our department and computer system.

I would like to thank most of all my parents for their support when I most needed it, and for their encouragement for me to always do my best.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A STUDY OF THE FUNCTIONAL PROGRAMMING LANGUAGE FP

Beverly P. Rollins, M.S.
Oregon Graduate Center, 1983

Supervising Professor: Richard B.Kieburtz

FP has been proposed as an alternative to Von Neumann and applicative languages, in which word-at-a-time thinking, and abstraction on variables obscure the problem that the programmer is attempting to solve.

Lazy, or delayed evaluation, allows the evaluator to avoid unnecessary evaluation, and allows the programmer to work with infinite objects. The benefits and detriments of lazy evaluation are discussed.

FP is also extended with a fixpoint functional and an apply function, and the affects of these additions to the language are discussed.

1. INTRODUCTION

John Backus [1] stated that conventional programming languages have become "fat and flabby", and that the many new features added to the languages add little to their power and obscure their semantics. Formal descriptions of these languages tend to be too bulky to be comprehensible, and hence are of little use. Since the languages are modelled after the von Neumann machine, they encourage the programmer to "word-at-a-time" thinking, instead of thinking in large conceptual units.

Backus' suggested solution to these problems is the concept of a *Functional Programming System* (FP System). A *function-level* program is a constructive presentation of a mathematical function. One builds new programs by combining existing programs, using functional forms such as composition and construction. A program is more easily understood if it is built from parts that are easily understood. Such a program is also more easily modified and verified.

Von Neumann and applicative programming, on the other hand, involve the programmer in lower-level details that can obscure the problem. In von Neumann-style programming one builds programs with assignments and control flow constructs, that operate on objects closely resembling the computer's storage cells. In applicative programming one builds a function by first building an expression that denotes an

object, and then abstracting on one or more variables within that expression. Both applicative and Von Neumann programming involve more complicated naming systems and substitution rules than does function-level programming.

Each of the following three examples is a program that takes as input two integer lists of equal length, and returns a list of integers, each member of which is the product of the corresponding members of the input lists. One can see in the Pascal program that the programmer had to deal with a counter variable and explicit iteration through arrays, details which have little to do with the basic solution to the problem. The Lisp example illustrates the kind of recursive program structure typical of Lisp programs. The FP program exhibits the use of composition to link two independent programs to create a new program. This method of constructing programs allows the programmer to think on a very high level, and also allows him to reason about and prove things about his programs. Backus [1] presents an algebra of programs that allows one to algebraically manipulate FP programs and prove theorems about them.

Example 1.1: A Pascal program that multiplies the members of two lists

```
type list = array[1..MAX] of integer;
function listmul(L1,L2 : list) : list;
var temp : list;
    i    : integer;
begin
for i := 1 to MAX do
  temp[i] := L1[i] * L2[i];
listmul := temp
end;
```

Example 1.2: A LISP program that multiplies the members of two lists

```
(def listmul
  (lambda (L1 L2)
    (cond ((null L1) nil)
          (t (cons (times (car L1) (car L2)) (listmul (cdr L1) (cdr L2)))))))
```

Example 1.3: An FP program that multiplies the members of two lists

$$\text{def listmul} = \alpha\text{mul} \circ \text{trans}$$

FP systems are still at the experimental stage, and much remains to be done before they are useful for substantial programming projects. The intent of this thesis is to study the FP language first proposed by Backus [1] as an example of programming at the function level. An FP interpreter was implemented with two major changes made to the language:

1) The interpreter follows a ***lazy*** (i.e., delayed) evaluation strategy. Lazy evaluation allows one to avoid unnecessary evaluation and to model infinite objects, but an efficiency penalty is paid.

2) The language is extended with a fixpoint operator and an ***apply*** function.

Running times of the lazy interpreter are compared with those of a strict interpreter, and a rationale for the results is given.

We show that the addition of ***apply*** makes FP significantly more expressive, but that, even with ***apply*** added, FP is still not as expressive as the lambda calculus.

Notation

There is some FP terminology which is used a great deal in examples, and also in expressions which do not belong to FP:

< ... >	- sequence brackets
[...]	- construction brackets (e.g. [f,g]:x = <f:x,g:x>)
∘	- function composition (e.g. f∘g:x = f:(g:x))
:	- application

$e_1 \rightarrow e_2 ; e_3$ - if e_1 then e_2 else e_3 (conditional)

ϕ - the empty sequence (<>)

$\bar{a}$ - the function that always returns a

other notation:

$\Rightarrow$ - evaluates to, in one step

$\overset{*}{\Rightarrow}$ - evaluates to, in one or more steps

The FP functions and functionals that are used in this paper are defined in Appendix B. A good overview of FP can be found in [1].

2. DESCRIPTION OF FP

The FP that we are studying is actually a member of the set of FP languages, or FP systems. An FP language consists of a set of objects, a set of functions, and a fixed set of functionals (or functional forms). Each version of FP is identified by the contents of the above domains. We have chosen to use the FP that Backus describes [1] with several modifications. Whenever we refer to "FP" we will be referring to our version of FP.

FP differs from Backus' system in the following points:

1) Real numbers are not included in the domain of atoms.

2) The empty sequence is not an atom.

3) Evaluation of some functions is lazy, allowing for infinite objects to have meaning. This means that "lazy" functions are not bottom-preserving.

The syntax of our FP is the same as Backus', except that the selectors 1,2,...,n are 1^{st},2^{nd},...,n^{th}, and literals (i.e., character strings) are surrounded by single quotes. The purpose of these changes is to eliminate any confusion that might exist between function names and literals (f vs. 'f'). These changes in syntax become necessary later on when we make extensions to FP that allow us to treat functions as objects, requiring that functions and objects be syntactically distinct. A concrete grammar for

FP is given in Appendix G.

2.1. The Difference between Functionals, Functions, and Objects

There are two algebras in FP: the algebra of objects, whose operators are functions that take objects to objects, and the algebra of functions whose operators are functional forms that take functions to functions. Unless otherwise specified, "function" will refer to an entity of type "object → object", and "functional" will refer to an entity of type "(object → object) → (object → object)", or "function → function".

An FP program is a function. In order to construct a program one either gives a primitive FP function, or one builds a new function by applying a predefined functional to already existing functions. The programmer is then working in the algebra of functions and functionals. It is important to note that

1) The only available functionals are those that are predefined in FP. There is no way to create new functionals, i.e., a function can not be construed as a functional.

2) A functional can not stand by itself as an FP program.

In order to obtain the result of executing a program p, one applies it to an object as follows: p:x , where x is an FP object. The application operator (:) appears only once in the applicative expression. Functions can occur only on the left side of the colon, and an expression denoting an object occurs on the right side. This convention is different from the

lambda calculus where any number of applications can occur within an applicative expression and in which functions and primitive objects are not distinguished.

We must note here that in the evaluation sequence

$$[f_1, \ldots, f_n]:x \Rightarrow \langle f_1:x, \ldots, f_n:x \rangle$$

the application $f_i:x$ actually means "the result of computing $f_i:x$", for each $1 \leq i \leq n$. The evaluation of the same expression when performed by the *lazy* interpreter would be:

$$[f_1, \ldots, f_n]:x \Rightarrow \langle\!\langle f_1:x, \ldots, f_n:x \rangle\!\rangle$$

$\langle\!\langle f_1:x, \ldots, f_n:x \rangle\!\rangle$ denotes a *suspension* in which $f_i:x$ for $1 \leq i \leq n$ is left unevaluated. (Suspensions will be discussed in Chapter 6, however we bring this up now to hopefully dispell some confusion that could arise concerning the placement of the application operator.) The suspension seems to contradict the rule of (:) placement because the application operator occurs more than once, however it is a result given only by the lazy interpreter, and can not be input legally by the user. Although the suspension contains functions and applications, it still *denotes* an object, and may occur on the right side of an application.

A clarification should also be made concerning the number of arguments given to a function. All FP functions (programs) have the type "object → object". When a function conceptually takes more than one argument as its input, we put all of its arguments into a sequence, and so it will map a single object argument (i.e., the sequence) into its result.

Sometimes for convenience it is implied that a function takes more than one argument, or its n^{th} argument is referred to. For example, when one states that *add* takes two numbers as its arguments, one actually means that *add* takes a single argument that is a sequence of two numbers.

2.2. The Domains of FP

An FP object is either a primitive object, i.e., an integer, character string (e.g., 'a','0X+'), or Boolean (T or F), or it is a sequence of objects (e.g., <1>, <<'a','b'>,'c'>). The empty sequence is denoted by <> or ϕ. There is one more object, $\perp$ (read as "bottom"), which denotes "undefined".

Let a *pair* be a sequence of two objects that are of the same type. The predefined functions of FP can be roughly divided into two groups: the arithmetic-logical functions and the functions that operate on sequences. The arithmetic-logical functions include such functions as *add*, *sub*, *mul*, which operate on integer pairs, and the logical functions *and*, *or*, *not*, which operate on Booleans and Boolean pairs. The sequence functions can be divided into those that build sequences (*apndl* (append on the left), *apndr* (append on the right)), and those that manipulate them (e.g., *distl* (distribute left); *trans* (transpose); the selectors 1^{st}, 2^{nd} ,...; *tl* (tail)). There remain polymorphic functions such as *id* (the identity function) that operate on arbitrary argument types.

The user can define new functions by combining previously defined functions using the functionals (or combining forms) that are provided by FP. The set of functionals is fixed; no new functionals may be defined by the user. The most commonly used functionals are:

construction - $[f1,\ldots,fn]:x \Rightarrow \langle f1:x,\ldots,fn:x\rangle$

conditional - $(f \rightarrow g\ ;\ h):x \Rightarrow$ if $f:x = T$ then $g:x$
else if $f:x = F$ then $h:x$
else $\perp$

composition - $f \circ g:x \Rightarrow f:(g:x)$

The only structured object in FP is the sequence. Therefore the functions that would usually operate only on tuples (e.g., the selectors) and the functions that would usually operate only on lists (e.g., *tl*) all operate on the same type of object: sequences. This loose typing allows such expressions to occur:

$3rd \circ apndl:\langle 1,\langle 2,3\rangle\rangle$

$tl \circ [f,g,h]:x$

This freedom makes static type checking in FP extremely difficult. For example, in order to compute the type of $3rd \circ apndl \circ f \circ g : x$ one must at least partially evaluate $f \circ g : x$ to obtain the type of the third element

of apndl ∘ f ∘ g:x.*

When FP is extended by adding the function *apply* (see section 6.1) it becomes necessary to include functions in the domain of objects. Here is an example evaluation sequence:

apply:<add,<1,2>> ⇒ add:<1,2> ⇒ 3

We see that a function has occurred on the right side of the application, and so the syntactic distinction between functions and objects has become blurred. *Apply* violates the premise that all FP functions have the type "object → object", and thus FP with *apply* added no longer belongs to the set of FP languages. We will call our extended language *AFP*, a member of the set of *AFP languages*. Later on we will show that, although *apply* is a higher-order function, it is neither an FP function nor an FP functional, but lies somewhere between the two.

---

* A solution to the problem of typing FP is proposed by Guttag, Horning, and Williams [5]. Another solution which has been proposed is simply to modify FP so that it is strongly typed.

3. THE INTERPRETER

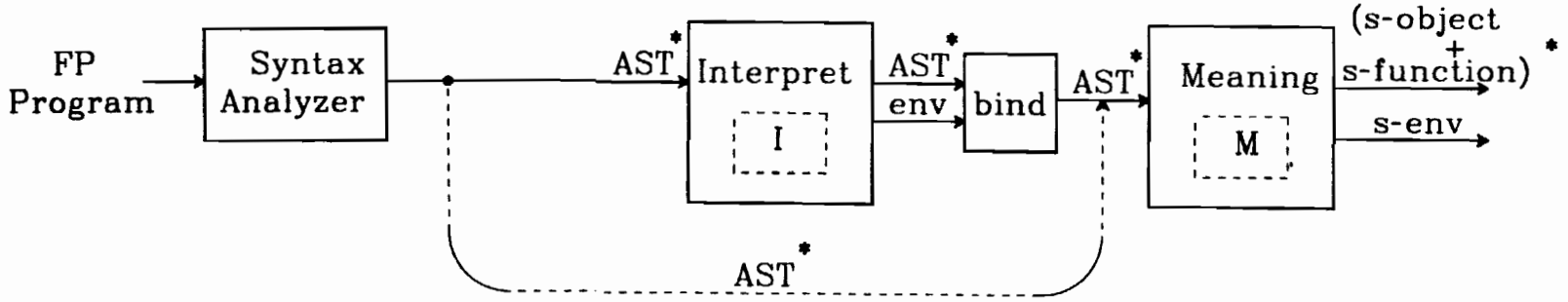
The principle function of the interpreter, *Interpret*, is written in Franz Lisp. It takes as input a list of Abstract Syntax Trees (ASTs) and a state, and outputs a list of reduced ASTs, an environment, and a state.

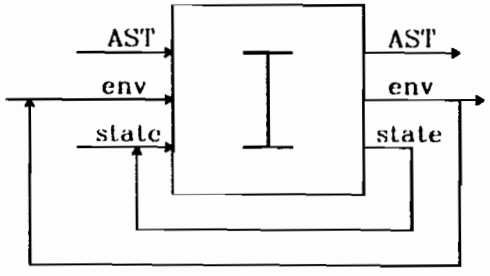
The list of ASTs is the output of submitting the user's FP expressions to a syntax analyzer. The environment used by *Interpret* is only needed to map user-defined function names to their definitions, and is therefore static. In order to avoid evaluating the same expression more than once, *Interpret* replaces an expression with the result of its evaluation. The state, which maps locations to AST's, is needed in order to express that replacement. The formal definition of the interpreter is in Appendix A.

The meaning of a list of FP ASTs is given by the function *Meaning*. *Meaning* maps a list of ASTs into a list of *s-objects* (semantic-objects) and *s-functions* (semantic-functions). The domains s-object and s-function are basically the objects and functions denoted by FP objects and functions. They are defined formally in Appendix B, along with the definition of *Meaning*.

The input that the user gives to the interpreter is a list of FP expressions, each of which is either a definition or an application. The Syntax Analyzer outputs a list of ASTs that denote the FP expressions. *Interpret* will take that list and return a list of reduced ASTs and an

environment. To obtain the meaning of a list of ASTs and their environment one *binds* all user-defined identifiers to their definitions and passes the resulting list to *Meaning*. (This is done so that *Meaning* will not have to use the environment that is defined within *Interpret*.) *Meaning* then returns a list of s-objects and s-functions. It is claimed that if one were to by-pass *Interpret* and send the AST-list directly to *Meaning*, the result would be the same as if *Meaning's* input was first sent through *Interpret* and *bind*, providing that the input is well-typed. In other words, *Interpret* preserves the meaning of its input. Figure 3.1 gives an over-all picture of the operation of the interpreter.





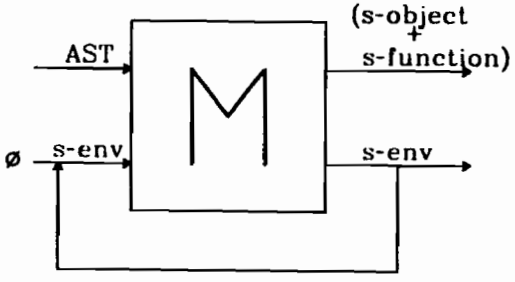


Figure 3.1 The Interpreter

4. THE INTERPRETER-LEVEL SOURCE LANGUAGE

The actual input given to the interpreter is in the form of abstract syntax trees. This representation is convenient to use as an intermediate language because it includes only semantically significant details. The translation from parse trees to abstract syntax trees has been formalized [9] , where the ***abstract syntax grammar*** is developed as a way of formally defining the structure of an abstract syntax tree language. A brief description of abstract syntax trees (ASTs) and abstract grammars follow. For a formal description see [9].

4.1. Abstract Syntax Trees

The abstract syntax of a program gives the operators and operands necessary to express its semantics. A convenient way to represent the abstract syntax is through abstract syntax trees (ASTs). The root of an AST represents an operator, and the subtrees represent its operands. Example FP ASTs and the FP programs from which they are derived are provided in Figures 4.1 and 4.2.

4.2. Abstract Syntax Grammar

An abstract syntax grammar is very similar to a context-free grammar, and gives us a way to formally define an abstract (AST) language. It consists of a set of equations and a set of productions.

Each equation defines a class of ASTs. Given equation $S = \alpha_1, \ldots, \alpha_n$, S is the name of a class of ASTs, and $\alpha_1, \ldots, \alpha_n$ are names of terminal or non-terminal nodes. Class S contains all ASTs whose root belongs to $\{\alpha_1, \ldots, \alpha_n\}$.

Each production defines the structure of all ASTs whose root name is the name on the left side of the production. The production $X \rightarrow S_1, \ldots, S_m$ tells us that an AST whose root node is X must have subtrees that belong to the classes $S_1, \ldots, S_m$, in that order. The right-hand side of a production may be a *regular expression* over class names. The regular expression is a shorthand for a (possibly infinite) series of productions.

Let α,β, and γ be regular expressions over class names in:

- "$A \rightarrow \alpha(\beta)^*\gamma$" represents the infinite number of productions: "$A \rightarrow \alpha\gamma$", "$A \rightarrow \alpha\beta\gamma$", "$A \rightarrow \alpha\beta\beta\gamma$" ... ($\alpha$ and γ may be empty)
- "$A \rightarrow \alpha|\beta$" represents the two productions "$A \rightarrow \alpha$" and "$A \rightarrow \beta$"

The abstract syntax grammar for FP follows, after which follow two examples of FP ASTs.

Abstract Syntax Grammar

Syntactic Domain Equations:

atom ={Numeral,Literal}

definition ={def}

application ={apply}

object ={atom,sequence,⊥}

f-expr = {condition, compose, select, Rselect, construct, constant, insertL, applyAll, bu, while, tl, id, atom, eq, gt, ge, lt, le, ne, null, reverse, distl, distr, length, add, sub, mul, div, trans, and, or, not, apndl, apndr, Identifier}

Productions:

program → (definition + application)*

def → Identifier f-expr

apply → f-expr (object + suspension + apply)

condition → f-expr f-expr f-expr

compose → f-expr f-expr

select → Numeral

Rselect → Numeral

construct → f-expr *

constant → object

insertL → f-expr

applyAll → f-expr

bu → f-expr object

while → f-expr f-expr

sequence → object *

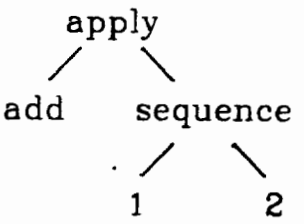


Figure 4.1 AST representation of add:<1,2>

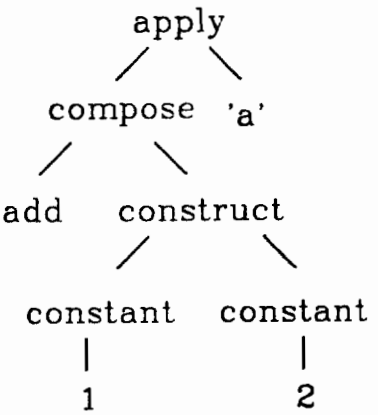


Figure 4.2 AST representation of add ∘ [$\bar{1}$,$\bar{2}$]:'a'

5. LAZY EVALUATION

Lazy evaluation, or delayed evaluation, has been compared to the call-by-name parameter passing of Algol [6] in that an expression is evaluated only when its value is needed. It differs from call-by-name in that once an expression has been evaluated it will never be re-evaluated (unless the same expression appears in two, unrelated places). Peter Henderson and James Morris [6,7] have presented lazy evaluation strategies for pure LISP in which arguments to functions are not evaluated until needed, and in which *cons* expressions are suspended until the parts of the expressions are demanded by another evaluation, or until the entire *cons* expression is explicitly forced. Arguments of a function are referenced by pointers so that any evaluation of the arguments is easily preserved. Because of the dynamic environments and bound variables of LISP, each suspension of an expression must preserve the environment in which the expression is to be evaluated.

Lazy evaluation in FP is like that of LISP with two exceptions:

1) Since there is only a single, static environment in FP, the environment does not need to be saved along with every suspension.

2) Since FP is not applicative, arguments to functions do not need to be suspended.

Lazy evaluation in FP requires that all list constructors be suspended. This allows the use of infinite lists, as in the following

example:

$$\textbf{def}\ \text{integers} = \text{apndl} \circ [\text{id}, \text{integers} \circ \text{succ}]$$

It also allows unnecessary evaluation to be avoided; the reduction

$$1^{st} \circ [f,g,h] : x \overset{*}{\Rightarrow} f:x$$

does not require evaluation of $g:x$ or $h:x$. Suspensions will be discussed later.

5.1. Lazy Interpretation of FP

5.1.1. Paradigm

Evaluation occurs on two levels: the functional level and the applicative level. All FP applications consist of one or more functions that are composed with each other and then applied to an object. Evaluation at the applicative level is carried out in strict, applicative (right-to-left) order, and occurs when a simple application, $f:x$, is being evaluated. (In a simple application the function contains no functionals.) The functional level refers to evaluation that does not need to know about the object at the end of an applicative expression, but instead manipulates the expression on the basis of the functions in that expression. Consider the application $f_1 \circ f_2 \circ \cdots \circ f_n : x$. The interpreter first operates on the functional

level. Evaluation on this level occurs left-to-right. Function f_1 is considered first. According to the value of f_1, one of six things can happen:

1) A value may be returned and further evaluation of the application will terminate. This only happens when f_1 is a constant function.

   Example: $\bar{1} \circ \text{add}:\text{'a'} \Rightarrow 1$

2) Evaluation of the rest of the application will be forced and evaluation will proceed at the applicative level. This happens when f_1 is an arithmetic-logic function which demands that its argument be completely evaluated (i.e. forced) to an object (i.e. when f is *strict*).

   Example: $\text{add} \circ [\bar{1},\bar{2}]:\text{'a'} \Rightarrow \text{add}:\langle 1,2\rangle$

3) The rest of the application ($f_2 \circ \cdots \circ f_n:x$) will be evaluated (evaluation always returns an object or a suspension) and evaluation will proceed on cases of that result. This occurs when f_1 is not a strict function, but needs to know something about its argument in order to proceed.

   Example:

$$\text{reverse} \circ 2^{nd} \circ [1^{st}, \text{apndl} \circ [1^{st}, 2^{nd}]] : \langle 1, \langle 2 \rangle \rangle$$
$$\Rightarrow \quad \text{reverse} \circ 2^{nd} : \langle\!| 1\text{st}:\langle 1,\langle 2\rangle\rangle, \text{apndl} \circ [1^{st}, 2^{nd}]:\langle 1,\langle 2\rangle\rangle |\!\rangle \qquad (3,6)$$
$$\Rightarrow \quad \text{reverse} \circ \text{apndl} \circ [1^{st}, 2^{nd}]:\langle 1,\langle 2\rangle\rangle \qquad (4)$$
$$\Rightarrow \quad \text{apndr} \circ [\text{reverse} \circ \text{tl}, 1^{st}] \circ [1^{st}, 2^{nd}] : \langle 1,\langle 2\rangle\rangle \qquad (5)$$

In the above example, *reverse* requires that the rest of the expression be evaluated, which it eventually is, to $\text{apndl} \circ [1^{st},2^{nd}]:\langle 1,\langle 2\rangle\rangle$. Then $\text{reverse} \circ \text{apndl}$ is transformed into $\text{apndr} \circ [\text{reverse} \circ \text{tl}, 1^{st}]$ according

to the definition of *reverse* in Appendix A.

4) Part of the rest of the application will be requested and evaluation will proceed at either level. This happens when f_1 is a selector, *tl* (tail), or *tlr* (right tail).

Example: $2^{nd} \circ [\overline{1},\overline{2}] : 'a' \stackrel{*}{\Rightarrow} \overline{2} : 'a'$

5) The composition $f_1 \circ \cdots \circ f_i$, $i \leq n$, is transformed into a new functional expression, and evaluation will proceed at the functional level. This happens for many of the non-strict functions. In Section 5.2 we show that the transformations employed by the interpreter preserve the meanings of the programs they operate on.

Example: $1^{st} \circ \text{apndl} \circ [\overline{1},\overline{\langle 2 \rangle}] : 'a' \Rightarrow 1^{st} \circ [\overline{1},\overline{\langle 2 \rangle}] : 'a'$

6) The entire expression is suspended. This happens when f_1 is one of the list constructors: *apndl*, *apndr*, or [...] .

Example: $\text{apndl} \circ g : x \Rightarrow \text{apndl} \circ g : x$

Example: $\text{apndr} \circ g : x \Rightarrow \text{apndr} \circ g : x$

Example: $[g_1, \ldots, g_m] : x \Rightarrow \langle\!\langle\, g_1{:}x, \ldots, g_m{:}x \,\rangle\!\rangle$ (The applications within the suspended construction are unevaluated.)

Evaluation of expressions containing functionals such as conditional, depend on the evaluation strategy for that functional. For example:

$(\mathbf{null} \rightarrow \mathrm{id}\ ;\ \mathrm{tl}) \circ \mathrm{apndl} \circ [1^{st}, \mathrm{id}] : \langle 1,2,3 \rangle$

$\mathbf{null} \circ \mathrm{apndl} \circ [1^{st}, \mathrm{id}] : \langle 1,2,3 \rangle$ (eval. of condition)

$\Rightarrow$ F

$\mathrm{tl} \circ \mathrm{apndl} \circ [1^{st}, \mathrm{id}] : \langle 1,2,3 \rangle$ (eval. of "false arm")

$\Rightarrow$ $2^{nd} \circ [1^{st}, \mathrm{id}] : \langle 1,2,3 \rangle$ (5)

$\Rightarrow$ $2^{nd} : \langle 1^{st} : \langle 1,2,3 \rangle, \mathrm{id} : \langle 1,2,3 \rangle \rangle$ (6)

$\Rightarrow$ $\mathrm{id} : \langle 1,2,3 \rangle$ (4)

$\Rightarrow$ $\langle 1,2,3 \rangle$

5.1.2. Suspensions

There are three kinds of suspensions, one for each of the sequence constructors: *apndl*, *apndr*, and [...] . Applications whose leftmost functions are *apndl* or *apndr* evaluate to themselves. This allows us to use infinite objects. For example, we can talk about the infinite list of integers and can extract part of that list without having to evaluate it completely:

def integers $\equiv$ apndl $\circ$ [id,integers $\circ$ succ]

$1^{st} \circ \mathrm{integers} : 1$

$\Rightarrow$ $1^{st} \circ \mathrm{apndl} \circ [\mathrm{id}, \mathrm{integers} \circ \mathrm{succ}] : 1$ (integers)

$\Rightarrow$ $1^{st} \circ [\mathrm{id}, \mathrm{integers} \circ \mathrm{succ}] : 1$ (Theorem T12*)

$\Rightarrow$ $1^{st} : \langle \mathrm{id}:1, \mathrm{integers} \circ \mathrm{succ}:1 \rangle$ ([...])

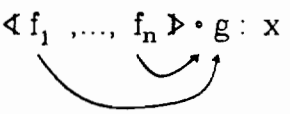
$\Rightarrow$ $\mathrm{id} : 1$ (1st: $\langle \cdots \rangle$)

$\Rightarrow$ 1 (id)

---

* The transformation $1^{st} \circ \mathrm{apndl} = 1^{st}$ is proven correct in Appendix C.

A construction such as $[f_1, \ldots, f_n] \circ g : x$ evaluates to $\langle f_1 \circ g:x, \ldots, f_n \circ g:x \rangle$. This is done in order to implement value sharing. Each member of the construction is made to point to the expression to the right of the construction, so we actually have:



which is written as

$$\langle f_1 \circ g:x, \ldots, f_n \circ g:x \rangle.$$

We suspend constructions so that unnecessary work may be avoided. Each application within the suspended construction is unevaluated until its value is requested. For example:

$$1^{st} \circ [f,g,h] \circ j:x \Rightarrow 1^{st}: \langle f \circ j:x, g \circ j:x, h \circ j:x \rangle$$
$$\Rightarrow f \circ j:x$$

The function *force* will evaluate its argument to an object, but will first evaluate* its argument so that an expression such as $force \circ 1^{st} \circ [\overline{1}, \underline{1}] : 'a'$ will be defined.

$force \circ 1^{st} \circ [\overline{1}, \underline{1}] : 'a'$
$\Rightarrow force \circ 1^{st}: \langle \overline{1} : 'a', \underline{1} : 'a' \rangle$
$\Rightarrow force \circ \overline{1} : 'a'$
$\Rightarrow force : 1$
$\Rightarrow 1$

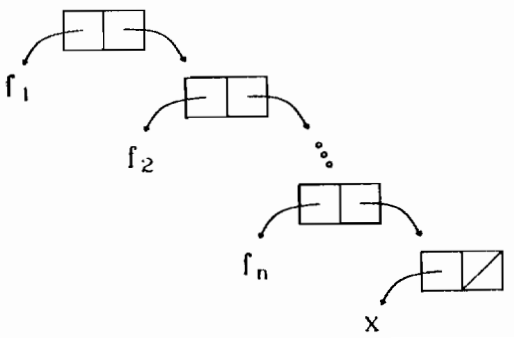
---

* Evaluation always results in an object or a suspension.

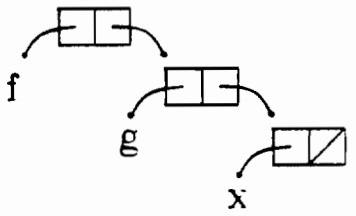
Note that the function $\overline{\perp}$, the function that returns bottom for every argument, is not a legal input by the user (neither is $\perp$, the object bottom). It was used in this example to give an obvious instance of a bottom-producing function, and might have been realized at the source level by a user-defined function whose evaluation would fail to terminate if it were applied to any argument.

5.1.3. Value Sharing and the FP Application Data Structure

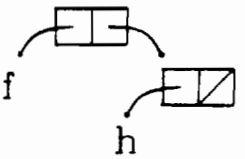
In order to avoid re-evaluating an expression once it has been evaluated we employ the following procedure: whenever an expression *e* is evaluated to *e'*, replace *e* with *e'*. This can be done very easily in LISP using a simple list and the LISP *replace* function. The FP expression $f_1 \circ \cdots \circ f_n : x$ is represented in LISP as $(f_1 \cdots f_n\ x)$, the LISP implementation of which actually looks like:



The LISP list structure lends itself well to the strategy of looking at the first item and replacing a rightmost section of the list with the result of its evaluation. Suppose that we are evaluating the expression f ∘ g : x. Suppose that function f demands that the rest of the expression, i.e., g : x, be evaluated. The LISP representation of f ∘ g : x is:



If g : x evaluates to h, then g : x will be replaced by h and we will have:



In order to implement value sharing in constructions we make each member of the suspended construction point to the part of the expression that follows the construction. Then when one member of the construction causes complete or partial evaluation of the rest of the expression, the result of the evaluation is reflected in all of the members of the suspension. We use the special brackets ◁ and ▷ to surround a suspended construction. The suspended construction has the type of a sequence object, however it still contains unevaluated applications. Therefore we have chosen the special brackets to distinguish it from both objects and functions. When the application of a construction is

evaluated, the resulting suspension will be the object of a further application in the consequent expression. For example:

$$f \circ [g,h] \circ i:x \Rightarrow f: \langle g \circ i:x, h \circ i:x \rangle$$

5.2. Correctness of Transformations

The principal evaluator function of the interpreter is:

eval : (object+application) → state → env → (object+application) × state.

If, in the definition of the interpreter, $\text{eval}[\![e]\!]\eta\sigma = \text{eval}[\![e']\!]\eta\sigma$, we say that e is transformed into e'. Lazy evaluation is only permitted under a scheme where e' has the same meaning as e *wherever e is defined*. e' may be defined on arguments where e is not defined. Therefore, when we are performing lazy evaluation, we only need to show that the meaning of e *approximates* the meaning of e'. The function

M : (object+application+definition+f-expr) → s-env →
(s-function+s-object) × s-env

gives the meaning of FP expressions, and is defined in Appendix B. We will ignore the environment (s-env), since it will not affect our discussion. For each transformation made by the interpreter we would like to show that

$$\text{eval}[\![e]\!]\eta\sigma = \text{eval}[\![e']\!]\eta\sigma \text{ implies } M(e) \sqsubseteq M(e').$$

Definition of $\sqsubseteq$:

$M(f) \sqsubseteq M(g) \iff M(f{:}x) = M(g{:}x) \;\forall\; x$ such that $M(f{:}x) \neq \perp$

Theorem 5.1: if $M(f) \sqsubseteq M(g)$, then $M(h \circ f \circ j) \sqsubseteq M(h \circ g \circ j)$

Proof:

Assumption: $M(f) \sqsubseteq M(g)$

Show: $M(h \circ f \circ j) \sqsubseteq M(h \circ g \circ j)$

1) $M(f \circ j{:}x) = M(f){:}(M(j){:}M(x))$	(composition)
2) if $M(f){:}(M(j){:}M(x)) \neq \perp$ then $M(f){:}(M(j){:}M(x)) = M(g){:}(M(j){:}M(x))$	($\sqsubseteq$)
3) and $M(f \circ j{:}x) = M(g \circ j{:}x)$	(composition)
4) $M(f \circ j) \sqsubseteq M(g \circ j)$	(1,2,3)
5) if $f{:}x \neq \perp$ then $M(f{:}x) = M(g{:}x)$	($\sqsubseteq$)
6) and $M(h){:}M(f{:}x) = M(h){:}M(g{:}x)$	(application)
7) and $M(h \circ f{:}x) = M(h \circ g{:}x)$	(composition)
8) $\therefore M(h \circ f) \sqsubseteq M(h \circ g)$	($\sqsubseteq$)
9) $M(h \circ (f \circ j)) \sqsubseteq M(h \circ (g \circ j))$	(assumption, 4 & 8)
10) $M(h \circ f \circ j) \sqsubseteq M(h \circ g \circ j)$	(associativity of $\circ$)

■

Corollary: if $M(f) \sqsubseteq M(g)$, and the FP function

$e = h_1 \circ \cdots \circ h_m \circ f \circ h_n \circ \cdots \circ h_p$ is transformed into

$e' = h_1 \circ \cdots \circ h_m \circ g \circ h_n \circ \cdots \circ h_p$, then $M(e) \sqsubseteq M(e')$.

According to the above corollary, we can make such transformations in any order and be assured that the resulting expression preserves the meaning of the expression we started with.

We demonstrate the correctness of the transformation tl∘apndr $\Rightarrow$ 2^{nd} below. The remainder of the transformations are proven correct in Appendix C. The environment and state arguments to *eval* and ***M*** have no effect on the proof, so we have taken the liberty of omitting them. For the sake of clarity we have also omitted some steps which should be obvious to the reader. For example:

M(tl:<a>) = M(tl):M(<a>) = (λ<O_1, . . . , O_n> .<O_2, . . . , O_n>):<a> = <>

has been abbreviated to M(tl:<a>) = <>. All FP objects are allowed to denote their meaning.

Theorem 5.2: M(tl∘apndl) $\sqsubseteq$ M(2^{nd})

M(tl∘apndl:x) $\neq \perp$ $\forall$ x suchthat apndl:x $\neq \perp$ and tl:(apndl:x) $\neq \perp$. It suffices for us to show that $\forall$ x such that apndl:x $\neq \perp$, M(tl∘apndl:x) = M(2^{nd}:x).

Proof:

Case 1: x = <a,<>>

M(tl∘apndl∘ <a,<>>) = M(tl):(M(apndl:<a,<>>))	(composition)
M(tl∘apndl: <a,<>>) = M(tl):<a>	(M(apndl))
= <>	(M(tl))
M(2^{nd}:<a,<>>) = <>	(M(2nd))

Case 2: $x = \langle a, \langle b_1, \ldots, b_n \rangle \rangle$

$M(tl \circ apndl:\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$

$= M(tl):M(apndl:\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (composition)

$= M(tl):\langle a, b_1, \ldots, b_n \rangle$ (M(apndl))

$= \langle b_1, \ldots, b_n \rangle$ (M(tl))

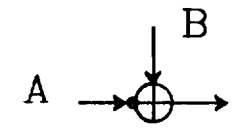
$M(2^{nd}:\langle a, \langle b_1, \ldots, b_n \rangle \rangle) = \langle b_1, \ldots, b_n \rangle$ ($M(2^{nd})$)

■

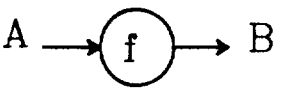
5.3. Process Networks

FP programs can model networks of communicating processes. In order to show this we use data-flow graphs to describe the programs. Data flow primitives are described in Table 5.1, and some examples of correlations between FP programs and data-flow diagrams are given in Table 5.2.

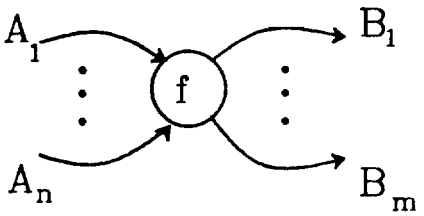
Table 5.1 Data Flow Primitives



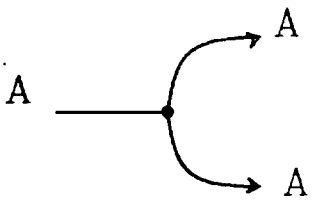
A is appended onto stream B.



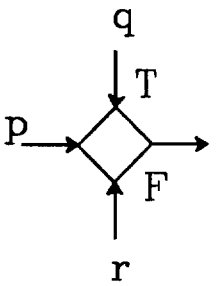
$f : A \rightarrow B$



$f : A_1 \times \cdots \times A_n \rightarrow B_1 \times \cdots \times B_m$

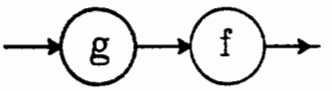
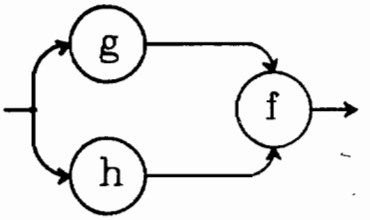
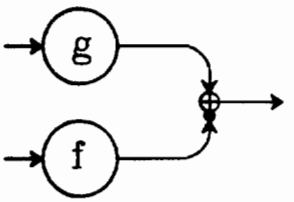
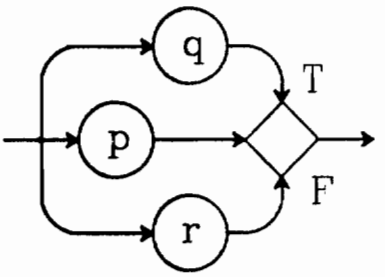


Two copies are made of A.

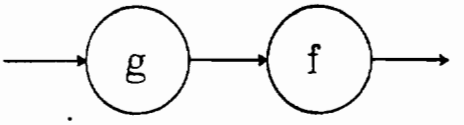


If p then q else r.

Table 5.2 Data Flow Modelling of FP Programs

FP PROGRAM	DATA FLOW DIAGRAM
f ∘ g	
f ∘ [g,h]	
apndl ∘ [f,g]	
(p →q; r)	

The composition $f \circ g$ describes a function which first sends its input to g, and then sends the output of g to f. It is very natural to model that function as:



If g sends a (possibly infinite) list, or stream, to f, then it is conceivable that g and f could operate in parallel. The transfer of information from g to f could operate in one of two ways:

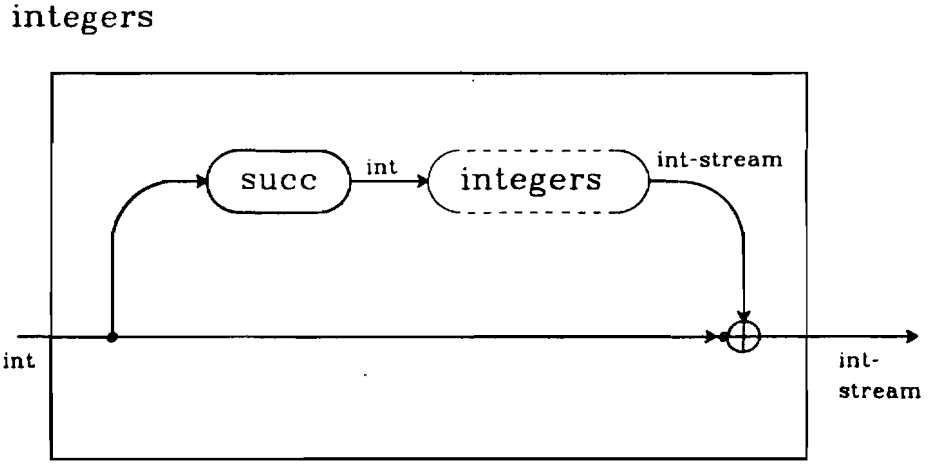
1) Function g sends its output (or pieces of it) to f whenever g is ready, and f processes its input as soon as it is available from g. This is a *data driven* operation and corresponds to standard applicative evaluation. When f and g work in data driven mode, they can each operate in parallel, depending only on the availability of their inputs.

2) Function g sends its output to f only when f makes a request for it. This is a *demand driven* operation, and corresponds to lazy evaluation. The overhead involved in this communication can be quite costly, as is demonstrated later in section 5.4.

A commonly used example of an infinite object defined in FP is the definition of the function *integers*.

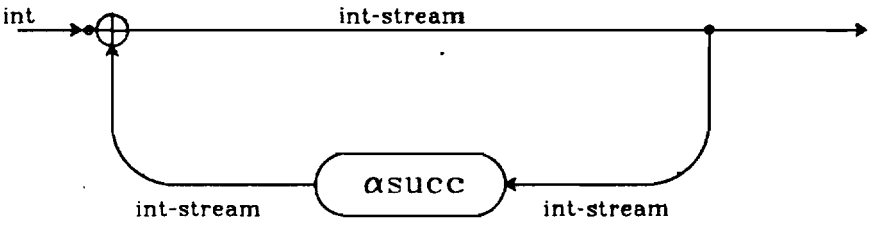
def integers = apndl ∘ [id,integers ∘ succ]

When we try to describe *integers* using a data flow graph, we encounter the problem of showing recursion in the graph. Henderson expresses recursion in a graph by using a special box to indicate the recursive call:



There is another definition of *integers*, however, in which the recursion can be diagrammed as iteration:

def *integers* = apndl ∘ [id, αsucc ∘ integers]



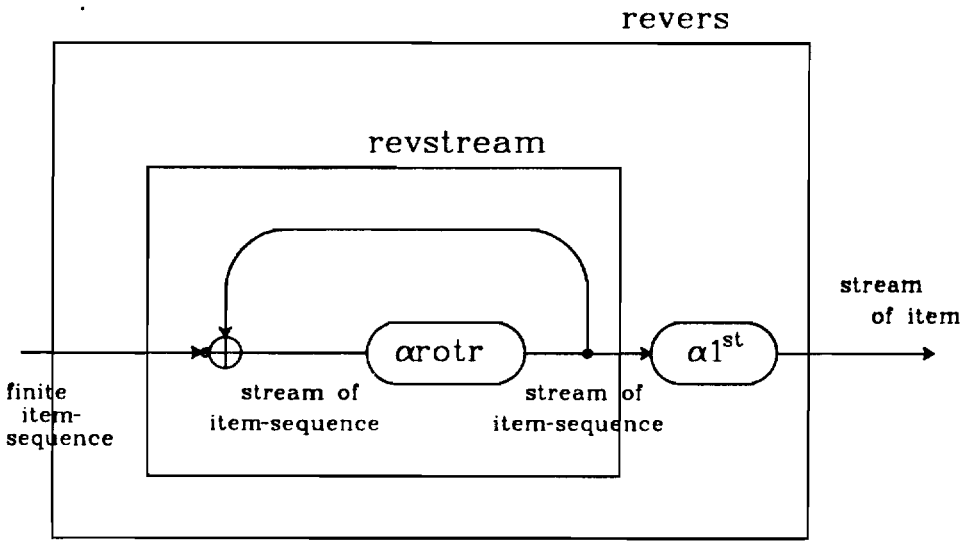
The latter example demonstrates a style of constructing programs in which the recursive call occurs as the right-most term in the composition. The applyall functional (α) tends to be very useful in this style.

Notice the difference between the types of the arcs in each graph. The arcs in the first graph hold single integers, and the recursive call causes the graph to iterate continually, thus producing an integer stream. The arcs in the second graph however hold streams of integers. Research is currently being done on this topic by Richard Kieburtz [8].

A data flow graph of a more complicated program follows. *Rotr* is the primitive FP function "rotate-right". α is the FP functional "apply-to-all", which is similar to the LISP *mapcar* function.

def revstream = αrotr $\circ$ apndl $\circ$ [id,revstream]

def revers = $\alpha 1^{st} \circ$ revstream



The function *revers* takes as input a finite sequence of items and returns as output an infinite stream of the same items in reverse order, repeating cyclically. For example, *revers* : <1,2,3> $\Rightarrow$ <3,2,1,3,2,1,3,2,...>.

5.4. Lazy Evaluation vs. Strict Evaluation

One of the "benefits" of lazy evaluation is that unnecessary work is avoided. Therefore it would be reasonable to assume that a lazy evaluator can perform calculations in less time than its strict counterpart takes to do the same calculations. This is not necessarily the case,

however, because the lazy evaluator may have to work very hard at being lazy.

We have run programs with both a lazy FP interpreter and a strict FP interpreter in order to compare running times. Both interpreters use the same data types, and have similar program structures, so that a reasonable comparison may be made.

The lazy interpreter evaluates the functions *distl*, *distr*, and *trans*, by doing source language substitution. (See Appendix A.) This is done so that expressions containing these functions may be transformed into expressions containing FP sequence constructors, and thus may be suspended. In the strict evaluator, however, these functions are *primitives*, i.e , they are directly interpreted in the host language, LISP. This makes a significant difference between the running times of the two interpreters. Therefore when we run programs using *distl*, *distr*, and *trans*, we also examine programs in which these functions are defined by the user. The first set of timings will show us what the strict evaluator can do when it can take advantage of the fact that *all* FP functions may be primitive because all arguments to the functions are objects and not suspensions. The second set of timings will give us a more equitable comparison.

5.4.1. Matrix Multiply

Matrix Multiply with System-Defined *distl*, *distr*, and *trans*

def IP ≡ /add ∘ αmul ∘ trans
def MM ≡ ααIP ∘ αdistl ∘ distr ∘ [1^{st},trans ∘ 2^{nd}]

Table 5.3 Running Times of Matrix Multiply (*MM*) with Strict and Lazy Evaluation

Program	Running Times †	
	Strict	Lazy
force ∘ MM:x*	12	143
1^{st} ∘ MM:x	19	10
force ∘ 1^{st} ∘ MM:x	13	62
1^{st} ∘ 1^{st} ∘ MM:x	11	17
force ∘ 1^{st} ∘ 1^{st} ∘ MM:x	13	17

* where x = <<<1,0,1>,<1,2,3>,<1,0,1>>,
<<1,1,1>,<0,1,1>,<1,0,1>>>

† unit is $\frac{1}{60}$sec.

Matrix Multiply with User-Defined *newdistl*, *newdistr*, and *newtrans*

def newdistl ≡ (null ∘ 2^{nd} → $\bar{\phi}$
; apndl ∘ [[1^{st},1^{st} ∘ 2^{nd}],newdistl ∘ [1^{st} ,tl ∘ 2^{nd}]])
def newdistr ≡ (null ∘ 1^{st} → $\bar{\phi}$
; apndl ∘ [[1^{st} ∘ 1^{st},2^{nd}],newdistr ∘ [tl ∘ 1^{st},2^{nd}]])
def newtrans ≡ (/ and ∘ αnull → $\bar{\phi}$; apndl ∘ [α1^{st},newtrans ∘ αtl])
def newIP ≡ / add ∘ αmul ∘ newtrans
def newMM ≡ ααnewIP ∘ αnewdistl ∘ newdistr ∘ [1^{st},newtrans ∘ 2^{nd}]

Table 5.4 Running Times of Modified Matrix Multiply (*newMM*) with Strict and Lazy Evaluation

Program	Running Times †	
	Strict	Lazy
force ∘ newMM : x*	74	141
1^{st} ∘ newMM : x	82	16
force ∘ 1^{st} ∘ newMM : x	76	53
1^{st} ∘ 1^{st} ∘ newMM : x	68	22
force ∘ 1^{st} ∘ 1^{st} ∘ newMM :	70	22

* where x = <<<1,0,1>,<1,2,3>,<1,0,1>>,
<<1,1,1>,<0,1,1>,<1,0,1>>>

† unit is $\frac{1}{60}$sec.

We see that the strict interpreter runs much faster than the lazy interpreter, except when part of MM:< ... > is requested. When $1^{st} \circ$ MM:$< \cdots >$ is requested, the lazy evaluator "beats" the strict evaluator. When $1^{st} \circ$ MM:$< \cdots >$ is *forced*, however, the lazy evaluator takes longer. In evaluating $1^{st} \circ 1^{st} \circ$ MM:$< \cdots >$, however, the lazy evaluator's times dramatically improve, and are even less than those of the strict evaluator when using *newdistl*, etc. This is because the lazy interpreter has limited itself to evaluating only $1^{st} \circ 1^{st} \circ$ MM:$< \cdots >$, a single inner product calculation, which requires very little work to evaluate. The strict interpreter first completely evaluates MM:$< \cdots >$ before taking $1^{st} \circ 1^{st}$ of it.

5.4.2. A Straight-Selection Sort

Straight-Selection Sort with System-Defined *distl*

def ls ≡ null ∘ 2^{nd} → $\overline{T}$; / and ∘ αlt ∘ distl
def least ≡ null → $\overline{0}$; ls ∘ [1^{st},tl] → 1^{st} ; least ∘ tl
def delete ≡ eq ∘ [1^{st},1^{st} ∘ 2^{nd}] → tl ∘ 2^{nd}
; apndl ∘ [1^{st} ∘ 2^{nd},delete ∘ [1^{st},tl ∘ 2^{nd}]]
def sort ≡ (null ∘ 2^{nd} → $\overline{\phi}$
; apndl ∘ [1^{st},sort ∘ delete ∘ [1^{st},2^{nd}]]) ∘ [least,id]

Table 5.5 Running Times of Straight-Selection Sort Using System-Defined *distl*, with Strict and Lazy Evaluation

Input to Evaluator	Running Times †	
	Strict	Lazy
force ∘ sort : <9,8,7,6,5,4,3,2,1,0>	153	710
1^{st} ∘ sort : <9,8,...,1,0>	153	138
force ∘ sort : <19,18,...,1,0>	771	5,121
1^{st} ∘ sort : <19,18,...,1,0>	767	612
force ∘ sort : <29,28,...,1,0>	2,139	16,602
1^{st} ∘ sort : <29,28,...,1,0>	2,101	1,395
force ∘ sort : <39,38,...,1,0>	4,842	43,288
1^{st} ∘ sort : <39,38,...,1,0>	4,420	2,521
5^{th} ∘ sort : <39,38,...,1,0>	4,579	12,420
10^{th} ∘ sort : <39,38,...,1,0>	4,599	23,373

† unit is $\frac{1}{60}$ sec.

Straight-Selection Sort with User-Defined *distl*

def newls ≡ null ∘ 2^{nd} → $\bar{T}$; / and ∘ αlt ∘ newdistl
def newleast ≡ null → $\bar{0}$; newls ∘ [1^{st},tl] → 1^{st} ; newleast ∘ tl
def newsort ≡ (null ∘ 2^{nd} → $\bar{\phi}$
; apndl ∘ [1^{st},newsort ∘ delete ∘ [1^{st},2^{nd}]]) ∘ [newleast,id]

Table 5.6 Running Times of Modified Straight-Selection Sort Using User-Defined *newdistl*, with Strict and Lazy Evaluation

Input to Evaluator	Running Times †	
	Strict	Lazy
force ∘ newsort: <9,8,7,6,5,4,3,2,1,0>	380	785
1^{st} ∘ newsort: <9,8,...,1,0>	399	149
force ∘ newsort: <19,18,...,1,0>	2,597	5,383
1^{st} ∘ newsort: <19,18,...,1,0>	2,734	688
force ∘ newsort: <29,28,...,1,0>	8,472	17,018
1^{st} ∘ newsort: <29,28,...,1,0>	8,227	1,468
force ∘ newsort: <39,38,...,1,0>	18,834	41,041
1^{st} ∘ newsort: <39,38,...,1,0>	18,928	2,625
5^{th} ∘ newsort: <39,38,...,1,0>	18,869	13,359
10^{th} ∘ newsort: <39,38,...,1,0>	19,248	23,216

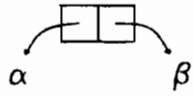
† unit is $\frac{1}{60}$ sec.

When using *distl*, and *forcing* the whole sorted list, the strict interpreter takes roughly ⅛ as long as the lazy interpreter. When evaluating the same programs using *newdistl* in place of *distl*, the strict evaluator takes roughly ½ as long. When evaluating 1^{st} ∘ sort: < · · · > or

$1^{st} \circ$ newsort: $< \cdots >$ the lazy interpreter always takes less time than the strict interpreter, and displays the most dramatic speed-up for $1^{st} \circ$ *new*sort: $< \cdots >$. When asked to return an item farther to the right in the sorted list, the lazy evaluator slows down. From evaluating $5^{th} \circ$ sort $\cdots$ and $10^{th} \circ$ sort $\cdots$ it is evident that a great deal more work is required for the lazy interpreter to extract more deeply nested items (i.e., items farther to the right) of the list. We propose an explanation for this in the next section.

5.5. Lazy FP vs. Lazy LISP

Friedman and Wise present a lazy implementation of LISP [4] in which evaluation of every *cons* expression is suspended as follows: (We will ignore the environment since it does not directly affect this discussion.)

eval(cons α β) = 

A "suspension node" is created whose fields each point to the (unevaluated) arguments of *cons*. Extracting part of a suspended *cons* is fairly straightforward.

FP suspensions are more complicated because there are *three* sequence constructors instead of one, and thus there must be three

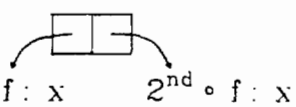
kinds of suspensions. The major problem, however, with FP suspensions, is that arguments are not readily available to the sequence constructors. This is because FP programs are built by *composing* functions, not by applying functions to their arguments. In order to suspend the expression $apndl \circ f : x$ in the same way that *cons* is suspended in LISP, we must evaluate $f : x$ until we obtain a pair.

One solution, which we have used in our interpreter, is to suspend all sequence constructor expressions, and to use transformations to extract parts of the suspensions. For example:

$$2^{nd} \circ \text{apndl} : x \Rightarrow 1^{st} \circ \text{tl} \circ \text{apndl} : x \Rightarrow 1^{st} \circ 2^{nd} : x$$

Another solution could be to transform an FP expression into an equivalent *applicative* expression, in which suspensions could be implemented as they are for LISP. We will present a technique for transforming a small subset of FP into an applicative form, and then will give an example evaluation sequence.

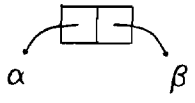
5.5.1. The Applicative Apndl Suspension

We will only consider a single sequence constructor, *apndl*. When we evaluate apndl ∘ f : x we want to return a suspension which we will denote by 

.

We will need functions to return the quantities pointed to by each of the fields of the suspension. We will call them *LEFT* and *RIGHT*. Finally, the

selector functions (we will only need 1^{st} for our example) will need to test their argument to see if it is a *suspension* or a *sequence*. If it is a sequence, a primitive selector function is called to operate on the sequence. If it is a suspension, *LEFT* and *RIGHT* and other selectors are used to delve into the suspension. For example:

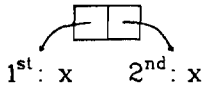
2^{nd}:  $\Rightarrow$ 1^{st}:(RIGHT: )

5.5.2. The Transformations

(l1) Let *LEFT* be a function that takes a suspension and returns what is pointed to by its first field.

(f1) Let *FUN* be a function that takes an FP functional expression and returns an equivalent function.

(f2) $FUN(f \circ g) = \lambda x\,.\,FUN(f):(FUN(g):x)$

(f3) $FUN(apndl) = \lambda x\,.$ 

(f4) $FUN(1^{st}) = \lambda x\,.$ x is a suspension → LEFT:x

; x is a sequence → FIRST:x

; ⊥

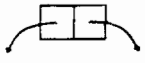
(e1) Let *EVAL* be a function that takes an FP application and returns either a suspension or an FP object. EVAL(f:x) = FUN(f):x

5.5.3. An Example

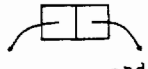
EVAL(1^{st} ∘ apndl: <1,<2>>)

= FUN(1^{st} ∘ apndl): <1,<2>> (e1)

= FUN(1^{st}): (FUN(apndl): <1,<2>>) (f2, beta red)

= FUN(1^{st}):  (f3, beta red)

1^{st}: <1,<2>> 2^{nd}: <1,<2>>

= LEFT:  (f4, beta red)

1^{st}: <1,<2>> 2^{nd}: <1,<2>>

= 1^{st}: <1,<2>> (l1)

= FIRST: <1,<2>> (f4, beta red)

= 1

If it turns out that lazy evaluation is more efficient on applicative languages, then this solution would allow one to program on the function level of FP, and then the implementation would be carried out on the applicative level. Note that the *strict* FP interpreter is implicitly applicative, since composition is defined in terms of application:

$$f \circ g:x \Rightarrow f:(g:x)$$

6. AFP AND YFP, EXTENSIONS OF FP

FP programs denote mathematical objects. They obey a set of laws that allow one to manipulate them and to reason about them in the very language that they are written in. These laws have a simple algebraic interpretation that allows them to be modelled easily, without recourse to a complicated denotational semantics. Ordinary reasoning by substitution of equals for equals provides a satisfactory logical framework for deducing properties of FP programs, without knowledge of sophisticated topics such as denotational semantics, logic, or axiomatic semantics.

One of the inadequacies of FP is that recursive functions can not be expressed without the aid of an environment. Therefore we propose the addition of a least fixpoint functional, Y. The functional Y expects its argument to be a function that takes a function and an object to an object. We will call this type of function a *scheme*. A scheme, written in FP, must denote the application of its first argument to its second. Therefore we will need to add an explicit *apply* function to FP when we add Y. Let AFP be FP with the addition of *apply*, and let YFP be AFP with the addition of Y. The definitions of *apply* and Y are included below. In the sections that follow we will compare FP to AFP, and in so doing will demonstrate the expressive power that is added to FP by *apply*. Then we will compare AFP to the lambda calculus, in order to show that, even with the proposed extensions, FP-languages are still less expressive than the

lambda calculus. Before we do that, however, we need to clarify the notion of *expressiveness*.

FP can clearly simulate a Turing machine, and so FP has the computing power of a Universal Machine. There are, however, some expressions that are directly representable in other Universal Languages that can not be *directly represented* in FP, but must be simulated. In this sense, FP is not as *expressive* as a language in which such expressions can be directly represented.

An expression *directly represents* another expression if their meanings are equivalent when computed in the same "computation environment", and when the *interpretation*, or decoding, of each object is always the same for the same object. The "computation environment" refers to all of the domains, such as environments or states, that are needed by the function that computes the meaning of an expression.

Now we give a definition of direct representation that is specific to the languages with which we are concerned. We use the function *Mu* to give meaning to expressions in AFP and APP.

$Mu : \text{AFP} \cup \text{APP} \cup \Lambda \rightarrow \Lambda$

$Mu(e) = \text{if } e \in \Lambda \text{ then } e$
$\quad \text{else if } e \in \text{AFP} \cup \text{APP then } u(e)$
$\quad \text{else } \perp$

The domains AFP and APP are defined later. Λ is the domain of lambda calculus expressions. The function μ is a (meaning) function that maps

AFP $\cup$ APP into Λ, and is also defined below. We will assume that the interpretation of an object never changes.

Definition: Let e_1 be an expression of language L_1 and let e_2 be an expression of language L_2 in

$$e_1 \textit{ directly represents } e_2 \text{ iff } Mu(e_1) \equiv_{\Lambda} Mu(e_2).$$

We have used the function *Mu* in order to show that an expression in the lambda calculus denotes its own meaning. Later on we will simply employ μ to show a direct relationship between languages when we know we will never have to compute the meaning of an expression in the lambda calculus, but are only interested in mapping AFP or APP to the lambda calculus. Similarly, on domains for which μ will only apply the function F, we will rely on F to give us the mapping.

6.1. Apply

$$\text{apply} = \lambda \langle f,O \rangle \,.\, f : O$$

The function *apply* takes a vector of two objects and returns the result of applying the first object to the second. Thus the first object is expected to be a function.

In order to add *apply* to FP then, we must include functions in the domain of objects. Formally, in the definition of the FP semantic function, we must include "s-function" in the domain of "s-object", i.e.,

$$\text{s-object} = \text{s-atom} + \text{s-sequence} + \perp + \text{s-function}$$

In the Abstract Grammar definition of the input to the interpreter, we must change the definition of "object" to:

$$\text{object} = \{\text{atom,sequence,}\perp\text{,f-expr}\}$$

The *apply* function allows us to write the sort of function schemes that, when given to Y, can be used to express a recursive function without explicit recursion.

6.1.1. The *Apply* Function and the Algebra of Programs

When *apply* is added to FP, the axioms for the algebra of FP programs remain consistent. This is demonstrated by substituting *apply* for function variables in the important distributive laws and showing that those laws are still consistent. We have included one example of this below, and the remainder of the proofs are in Appendix D.

Theorem 6.1: [f,g] ∘ apply = [f ∘ apply,g ∘ apply]

Proof:

[f,g] ∘ apply:<h,y>
⇒ [f,g]:(apply:<h,y>) (comp)
⇒ [f,g]:(h:y) (apply)
⇒ <f:(h:y),g:(h:y)> (constr)

[f ∘ apply,g ∘ apply]:<h,y>
⇒ <f ∘ apply:<h,y>,g ∘ apply:<h,y>> (constr)
⇒ <f:(apply:<h,y>),g:(apply:<h,y>)> (comp)
⇒ <f:(h:y),g:(h:y)> (apply)

■

6.1.2. The Power of *apply*

For an applicative expression such as f(g(h(x))) there exists an FP function whose meaning is equivalent, namely f ∘ g ∘ h : x. A series of applications that associate to the right can be expressed in FP by a series of compositions terminating with an application. There is, however, no way in FP to directly represent an applicative expression that associates to the left. In AFP, however, there exist expressions that can directly represent applicative expressions that associate to the left. For example:

apply ∘ [f,$\bar{x}$]:g
⇒ apply:<f:g,$\bar{x}$:g> (comp & constr)
⇒ apply:<f:g,x> (constant)
⇒ (f:g):x (apply)

We will define a set of languages, called APP-languages. Besides all of the objects, functional expressions, and single, well-typed applications of an FP-language (without *apply*), an APP-language contains all well-typed multiple applications of its members. (f:x) is well-typed if the type of f is t1 → t2, the type of x is t3, and t3 is a sub-type of t1. We will show that, given an APP-language, there exists an AFP-language that will directly describe it, whereas it cannot be directly described by an FP-language.

Let an AFP-language be an FP-language that includes *apply*, and in which functions *are* included in the domain of objects. Given an AFP-language L_{AFP}, let L_{APP} be the language of expressions "e" such that

1) e $\in L_{AFP}$

2) e = (c : d) and c $\in L_{APP}$ and d $\in L_{APP}$

3) Nothing else is in L_{APP}.

There exist two identity mappings Idf and Ida such that Idf takes all expressions of L_{AFP} into their syntactically identical counterparts in L_{APP}, and Ida = Idf^{-1} .

μ is a semantic function that maps all elements of L_{AFP} and L_{APP} into expressions of the lambda calculus. μ is defined in terms of the FP semantic function

F : f-expr → s-env → (s-object → s-object)

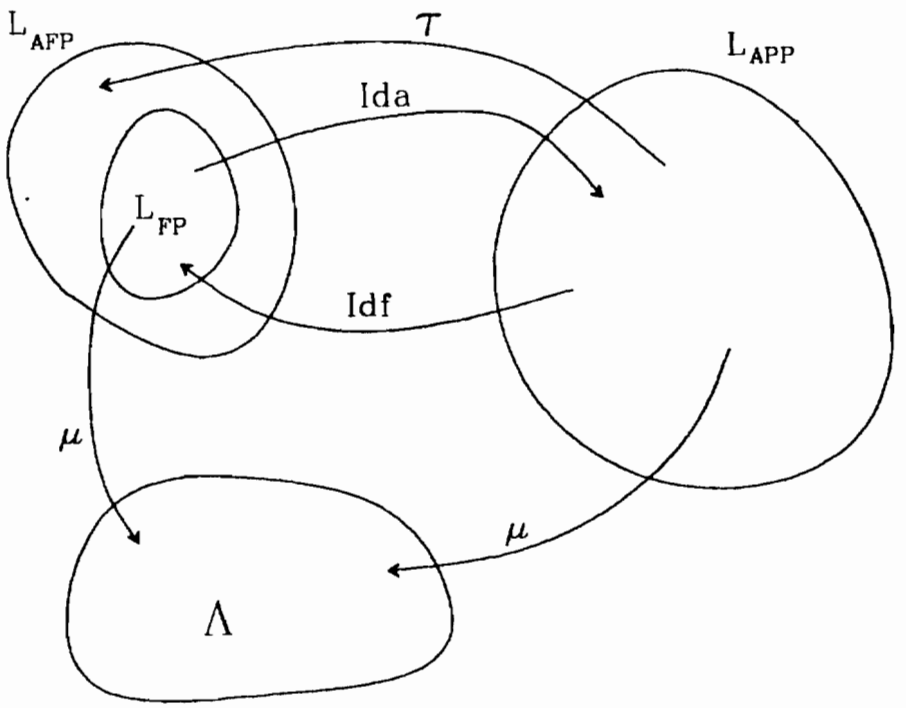
which is defined in Appendix B and gives the meanings of FP functions.

definition of μ:

(m1) $\mu(0) = 0_\sigma$ (0_σ is the object denoted by 0)
(m2) $\mu(a{:}b) = \mu(a){:}\mu(b)$
(m3) $\mu(f \circ g) = \lambda 0 . \mu(f){:}(\mu(g){:}0)$
(m4) $\mu([f_1, \ldots, f_2]) = \lambda 0 . \langle \mu(f_1){:}0, \ldots, \mu(f_n){:}0 \rangle$
(m5) $\mu(\text{apply}) = \lambda \langle f,x \rangle . f{:}x$
(m6) $\mu(f) = F(f)\phi$ (f is a functional expression)

Let L_{FP} be the FP language that is identical to L_{AFP}, except that the extensions needed for *apply* are missing. L_{FP} is a proper subset of L_{AFP}.

Claim: There exists a mapping, τ, that takes all of L_{APP} into L_{AFP} such that for all "a" in L_{APP}, $\mu(a) = \mu(\tau(a))$. (proof is in Appendix E)



τ uses the function: *Const* : $L_{AFP} \rightarrow L_{AFP}$. Intuitively, *Const* takes as input an object and returns a function that will return that object on any input. In other words, *Const* behaves like the "K" combinator. For simplicity, let O be the set of all members of $L_{APP} \cup L_{AFP}$ that are objects or functional expressions, and therefore contain no applications.

definition of τ:

(b1) $\tau(a) = a \quad (a \in O)$
(b2) $\tau(a:b) = a:b \quad (a,b \in O)$
(b3) $\tau(a:(b:c)) = \tau(a \circ b:c) \quad (a,b \in O)$
(b4) $\tau(a:(b:c)) = \tau(a:\tau(b:c)) \quad (a \in O, b \notin O)$
(b5) $\tau((a:b):c) = \tau(\text{apply} \circ [a,\text{Const}(\tau(c))]:b) \; (a \in O)$
(b6) $\tau((a:b):c) = \tau(\tau(a:b):c) \quad (a \notin O)$

(c1) $\text{Const}(a) = \bar{a} \quad (a \in O)$
(c2) $\text{Const}(a:b) = a \circ \text{Const}(b) \quad (a \in O)$

Example: evaluation of $\tau((a:(b:c)):(d:e))$ (a,b,c,d,e are functional expressions)

$\tau((a:(b:c)):(d:e))$

$= \tau(\text{apply} \circ [a,\text{Const}(\tau(d:e))]:(b:c))$	(b5)
$= \tau(\text{apply} \circ [a,\text{Const}(d:e)]:(b:c))$	(b2)
$= \tau(\text{apply} \circ [a,d \circ \bar{e}]:(b:c))$	(c2 & c1)
$= \tau(\text{apply} \circ [a,d \circ \bar{e}] \circ b:c)$	(b3)
$= \text{apply} \circ [a,d \circ \bar{e}] \circ b:c$	(b2)

Claim: τ preserves the meaning of (a:(b:c)):(d:e).

Proof:

μ((a:(b:c)):(d:e))
$=\mu$(a:(b:c)):μ(d:e) (m2)
$=(\mu$(a):μ(b:c)):μ(d):μ(e) (m2)
$=(\mu$(a):(μ(b):μ(c))):μ(d):μ(e) (m2)

μ(*apply*∘[a,d∘$\bar{e}$]∘b:c)
$=\mu$(apply∘[a,d∘$\bar{e}$]∘b):μ(c) (m2)
$=\mu$(apply):(μ([a,d∘$\bar{e}$]):(μ(b):μ(c))) (m2)
$=\mu$(apply):(λO.<μ(a):O,μ(d∘$\bar{e}$):O>):(μ(b):μ(c)) (m4)
$=\mu$(apply):<μ(a:(μ(b):μ(c))),μ(d∘$\bar{e}$):(μ(b):μ(c))> (beta red)
$=\mu$(apply):<μ(a):(μ(b):μ(c)),(λO∘μ(d):(μ(c):O)):(μ(b):μ(c))> (m3)
$=\mu$(apply):<μ(a):(μ(b):μ(c)),(λO∘μ(d):((λO∘μ(e)):O)):(μ(b):μ(c))>(m6)
$=\mu$(apply):<μ(a):(μ(b):μ(c)),μ(d):μ(e)> (beta red)
$=$(λ<f,x> . f:x):<μ(a):(μ(b):μ(c)),μ(d):μ(e)> (m5)
$=(\mu$(a):μ(b):μ(c))):(μ(d):μ(e)) (beta red)
$=\mu$((a:(b:c)):(d:e))

■

Function τ is a correspondence between L_{APP} and L_{AFP}, such that τ takes any member of L_{APP} to L_{AFP}. However, there exist elements of L_{APP} that can not be mapped to L_{FP} by τ. In other words, for any applicative expression that is built from FP functions, objects, and any number of nested and iterated applications, there exists an equivalent expression in an FP-language when it is extended by *apply*. (Recall that applicative expressions in an FP-language have only a single occurrence of the application operator (:).) Therefore *apply* enables FP (i.e., AFP) to directly express multiple applications of arbitrary nesting using a single application.

6.1.3. Is *apply* a Functional?

Recall that an FP functional maps FP functions to FP functions. *Apply* operates on functions, but can not be classified as a functional for the following reasons:

1) *Apply* can stand by itself as an FP program.

2) *Apply* maps function and object pairs to objects as well as functions. Functionals only map functions to functions.

3) *Apply* conforms to the syntax of FP functions.

Apply is not a functional, but is nevertheless significantly more powerful than ordinary FP functions. We can *almost* build new functionals with *apply*. *Apply* gives us the power to model arbitrarily nested applications, and allows us to build higher order functions, but does not allow us to build curried functions. For example, let us define the function *compose-apply*. *Compose-apply* is similar to the functional "∘", except that it takes three arguments instead of two.

def compose-apply ≡ apply ∘ [1^{st},apply ∘ [2^{nd},3^{rd}]]

compose : <f,g,x>

⇒ apply ∘ [1^{st},apply ∘ [2^{nd},3^{rd}]]: <f,g,x>	(compose-apply)
⇒ apply : <1^{st}:<f,g,x>,apply ∘ [2^{nd},3^{rd}]:<f,g,x>>	([], ∘)
⇒ apply : <f,apply:<2^{nd}:<f,g,x>,3^{rd}:<f,g,x>>	(1^{st},[], ∘)
⇒ apply : <f,apply:<g,x>>	(2^{nd},3^{rd})
⇒ **f** : (g : x)	(apply)

Apply gives us the ability to create functions such as *compose-apply*, but does not add to the *program-building* power of FP systems in the same way that the ability to create new functionals would. However, we must recognize that while *apply* is neither a functional nor an ordinary FP function, its addition to an FP language increases its expressiveness in a significant way. Such an extended FP language no longer falls into the category of "FP languages", and we have invented the category of "AFP languages" in which to place our "orphan" FP. Questions that should be answered concerning AFP languages are:

1) Are AFP languages significantly more *useful* than FP languages?

2) Are programs written in AFP languages easy to understand and manipulate?

6.2. The Least Fixpoint Functional

In an applicative language in which functions can be curried, a definition of the fixpoint function Y_{app} could be:

$$Y_{app}f \equiv \lambda O.((f:\bar{I}):O \neq \perp \rightarrow (f:\bar{I}):O\ ;\ (f:(Y_{app}f)):O)$$

Intuitively, $(Y_{app}f):O$, for a scheme f and an object O, is equivalent to $(f^n:(\perp)):O$ for the least integral value of n for which $(f^n:(\perp)):O$ is defined.

Since we cannot curry FP functions however (see section 6.3), f must take its arguments as a pair.

Y_{fp} is a functional in YFP, and therefore does not by itself denote a program. But coupled with a function f it has the following meaning:

$$Y_{fp}f \equiv \lambda 0.(f{:}\langle \bar{1},0\rangle \neq \perp \rightarrow f{:}\langle \bar{1},0\rangle \; ; \; f{:}\langle Y_{fp}f,0\rangle)$$

We will call the applicative fixpoint operator Y_{app}, and the YFP fixpoint operator Y_{fp}. Note that the type of Y_{fp} is different from the type of Y_{app}:

Y_{app} : $scheme_{app} \rightarrow function_{app}$

where $scheme_{app}$ = $function_{app} \rightarrow function_{app}$

and $function_{app}$ = object → object

Y_{fp} : $scheme_{fp} \rightarrow function_{fp}$

where $scheme_{fp}$ = $(function_{fp} \times object) \rightarrow object$

and $function_{fp}$ = object → object

About Y_{app} we can say:

$$f(Y_{app}f) = Y_{app}f$$

A law that we can state about Y_{fp} is not as intuitive:

$$Y_{fp}f = f \circ \overline{[Y_{fp}f, id]}$$

When we apply each side of the law to an object we get:

$$Y_{fp}f{:}x = f{:}\langle Y_{fp}f, x\rangle$$

which reflects the meaning of the fixpoint perhaps a little more clearly.

Y_{fp} is used to define recursive functions without having to explicitly use recursion. The function (scheme) f is expected to take a vector consisting of a function and an object, to an object. When $Y_{fp}f$ is applied to an object O, f is applied just as many times as is necessary to return an answer. The evaluation of factorial:3 (see below) requires that factscheme be called 4 times. The evaluation of factorial:4 would require 5 calls to factscheme, etc. Let $Y = Y_{fp}$ in the following example.

def pred = sub ∘ [id,$\bar{1}$]

def factscheme = eq0 ∘ 2^{nd} →$\bar{1}$

; mul ∘ [2^{nd},apply∘[1^{st},pred∘2^{nd}]]

def factorial = Y factscheme

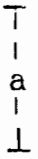
An example evaluation sequence:

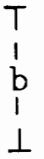
```
factorial : 3
Y factscheme : 3
factscheme : <Y factscheme, 3>
mul : <3, Y factscheme : 2>
          factscheme : <Y factscheme, 2>
          mul : <2, Y factscheme : 1>
                    factscheme : <Y factscheme, 1>
                    mul : <1, Y factscheme : 0>
                              factscheme : < 1̄ , 0>
                              1
                    1
          2
6
```

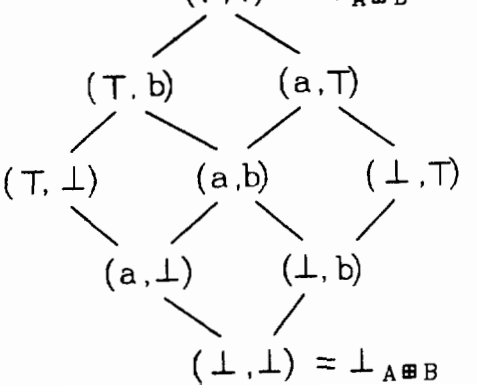
Given a function that never terminates, or a particular input for which it does not terminate, $\Upsilon f : 0$ computes forever. That would be the case in the example below if *succ* (the successor function) were substituted for ***pred*** in the definition of factscheme, or if factorial were applied to a negative number.

6.2.1. Domain Structure

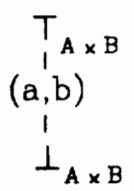
Note that adding Υ_{fp} as we have done requires that the Cartesian product domains of YFP be *separated* rather than *coalesced*, as is usually the case.

i.e., given domains A: 

B: 

separated product A ⊠ B : 

coalesced product A × B :



The choice of separated product implies that construction is non-strict. Our definition of Y_{fp} requires that f:<$\bar{1}$,a> not always result in ⊥, and this would not be possible in a semantics in which the product is strict, i.e., where <$\bar{1}$,a> = ⊥.

6.2.2. Y and the Algebra of Programs

As in section 6.1.1 it is easy to show that the addition of Y_{fp} does not disrupt the algebra of programs. The proof of this is in appendix D.

6.3. A comparison of AFP and the Lambda Calculus

In the following discussion Λ will refer to the domain of all lambda expressions. Let the set of AFP languages contain the set of FP languages plus *apply*.

There is a mapping from AFP programs to Λ. That mapping is the semantic function

F : f-expr → environment → lambda-expression

that is defined in the appendix. We will show that the range of *F* is a proper subset of Λ, and we will prove two properties that

characterize this subset. From this we will be able to conclude that any lambda expression that does not possess these properties is outside of this subset, and therefore can not be derived from an AFP program by F i.e., there exist some functions in the lambda calculus that cannot be directly represented in AFP.

The intent of this discussion is to demonstrate in at least one way how AFP is less expressive than the lambda calculus, and specifically to show that the higher order functions "K" and "Curry" are undefineable in AFP.

Definition: s is an *abstracted sub-expression* of e if s = λx.i and either

1) e = s

2) e = sa

3) e = as

4) s is an abstracted sub-expression of r, and λy.r is an abstracted sub-expression of e

Theorem 6.2: Every AFP program f is mapped by F to l in Λ such that l = λx.e and:

1) x does not occur free in any abstracted sub-expression of e

2) no variable occurs free in l

Proof: This can be shown by induction on the structure of AFP functions. By the definition of F in the appendix B and by the definition of *apply*, all predefined AFP functions satisfy the above properties. If the functions $f_1, \ldots, f_n$ satisfy the properties, then any function built from an FP combining form and any of the functions $f_1, \ldots, f_n$ also satisfies the properties because no combining form introduces any free variables.

Corollary: AFP is less expressive than the lambda calculus, and all of AFP can be directly described by a proper subset of the lambda calculus. In particular, it is impossible to express in AFP the equivalent of $\lambda x. \ldots \lambda y.e(x) \ldots$, where $\lambda x. \ldots \lambda y.e(x) \ldots$ is a lambda expression that contains abstracted sub-expression $\lambda y.e(x)$.

What if we wish to define the K combinator in AFP? In the semantic definition for K: $K = \lambda x.\lambda y.x$, x occurs free in the abstracted sub-expression $\lambda y.x$, so K does not satisfy the conditions stated above, and therefore cannot be directly defined within AFP. The same is true of the definition Curry $f = \lambda x.\lambda y.\ f:\langle x,y_1, \ldots, y_n\rangle$ where $y = \langle y_1, \ldots, y_n\rangle$. This is not to deny the possibility of "hardwiring" Curry and K into AFP. In fact it might be very desirable to increase the expressiveness of AFP by adding these

functions. But it would then become necessary to insure that such an expansion of the language does not change the properties that make it so useful to program and reason in.

7. CONCLUSION

We have defined and implemented a version of FP very similar to that given by Backus [1]. We have extended FP with Y and *apply*, and have shown that these extensions make FP significantly more powerful, but still less powerful than the lambda calculus.

We studied lazy evaluation of FP, and showed that while lazy evaluation can take less time than strict evaluation in certain circumstances, the overhead involved often increases running time greatly. Lazy evaluation of FP is complicated by the existence of *three* sequence constructors, and by the fact that composition and not application is the principal operator of FP.

FP's functional structure allows one to program at a higher level than one would using an applicative language. FP can also model data flows and networks very nicely. It's algebraic properties allow the programmer to manipulate and reason about his programs in a straightforward way.

It is the opinion of the author that FP has some serious problems. An FP that could be easily statically typed would be easier to program in. Although not a topic of this thesis, it is likely that it would be easier to perform lazy evaluation on a statically typeable FP. Lazy evaluation, although it shows promise, can be extremely costly. The addition of *apply* is necessary for a fixpoint functional, but adds just enough

applicative power to make one desire the ability to define curried functions. FP lacks the variety of data types needed for a comprehensive programming environment. But its clear, algebraic structure is nonetheless a very attractive medium in which to build programs.

REFERENCES

1. Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Comm. ACM* 21,8 (August 1978), pp. 613-641.

2. Backus, J. "Function-level computing", *IEEE spectrum*, August 1982, pp. 22-27.

3. Backus, J. "Function Level Programs as Mathematical Objects", In Proc. of 1981 Conference on Functional Programming Languages and Competer Architecture, ACM, New York, 1981, pp. 1-10.

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

5. Guttag, J., Horning, J., and Williams, J. "FP with Data Abstraction and Strong Typing", In Proc. of 1981 Conference on Functional Programming Languages and Competer Architecture, ACM, New York, 1981, pp. 11-24.

6. Henderson, P. and Morris, J., A lazy evaluator, In Proc. of 1976 Sympos. on Principals of Programming Languages, ACM, New York, 1976.

7. Henderson, P. *Functional Programming Application and Implementation*, Prentice-Hall International, London, 1980, pp. 231-239.

8. Kieburtz, R., "Marigold - A functional, flow-graph language", CS/E-82-08, Oregon Graduate Center, Beaverton, OR (August 1983).

9. Rollins, G., "Abstract Syntax in an Initial-Algebra Framework for Compilers", PhD Thesis, State University of New York, Stony Brook (1983).

APPENDIX A

FORMAL DEFINITION OF THE INTERPRETER

The Domains of Interpret

notation: a.l means "item a appended onto list l"

e{e'} means "environment e updated by environment e'"

env = Identifier → f-expr

state = AST → AST

The domains application, definition, f-expr, object and identifier, are subsets of the domain of AST's as defined by the Abstract Grammar.

The Interpretation Functions

Interpret : (application + definition)* → state → env → (AST* × env)

I : (application + definition) → state → env → (AST × state × env)

eval : (object + application) → state → env

→ ((object + application) × state)

defineable : (Identifier × f−expr × env) → boolean

predefined : Identifier → boolean

Fn : Identifier → (object → object)

FORCE : (object + application) → state → env → (object × state)

$$\text{Interpret}(t_1, \ldots, t_n)\sigma\eta \equiv \text{let } (r,\sigma_1,\eta_1) = I[\![t_1]\!]\sigma\eta \text{ in}$$
$$\text{let}(L,\eta_2) = \text{Interpret}[\![(t_2, \ldots, t_n)]\!]\sigma_1\eta_1 \text{ in}$$
$$n = 0 \rightarrow ((),\eta) ; (r.L,\eta_1\{\eta_2\})$$

$$I[\![\text{def i} \equiv \text{f}]\!]\sigma\eta \equiv \text{defineable(i,f,}\eta) \rightarrow (\text{i},\sigma,\eta[\text{f}/\text{i}]) ; (\perp,\sigma,\eta)$$
$$I[\![\text{f:x}]\!]\sigma\eta \equiv \text{let}(r',\sigma') = \text{eval}[\![\text{f:x}]\!]\sigma\eta \text{ in } (r',\sigma',\eta)$$

$$\text{defineable(i,f,}\eta) \equiv \eta(\text{i}) = \perp \rightarrow (\forall \text{x,identifier in f}),\ \eta(\text{x}) \neq \perp \text{ or x} = \text{i}$$
$$\rightarrow \text{true ;false}$$
$$; \text{false}$$

predefined(f) ≡ f ∈ {add,and,apndl,apndr,atom,distl,distr,div,
eq,ge,gt,id,le,length,lt,mul,ne,not,null,or,reverse,rotl,
rotr,sub,tl,tlr,trans}

Fn(Id) = cases Id is
add ⇒ ADD
and ⇒ AND
apndl ⇒ APNDL
apndr ⇒ APNDR
atom ⇒ ATOM
distl ⇒ DISTL
distr ⇒ DISTR
div ⇒ DIV
eq ⇒ EQ
ge ⇒ GE
gt ⇒ GT
id ⇒ ID
le ⇒ LE
lt ⇒ LT
length ⇒ LENGTH
lt ⇒ LT
mul ⇒ MUL
ne ⇒ NE
not ⇒ NOT
null ⇒ NULL
or ⇒ OR
reverse ⇒ REVERSE
rotl ⇒ ROTL

```
            rotr  ⇒  ROTR
            sub   ⇒  SUB
            tl    ⇒  TL
            tlr   ⇒  TLR
            trans ⇒  TRANS

eval(t)ση ≡ cases t is

    "an object" ⇒ (t,σ)

    force:x ⇒ (x,σ[x/t])

    Id:x ⇒ predefined(Id) → (r,σ[r/t]) where r = (Fn Id)x
                          ; let (r1,σ1) = eval⟦(η Id):x⟧ση in
                                (r1,σ1[r1/t])

    (p → f ; g):x ⇒ let (r1,σ1) = eval⟦p:x⟧ση in
                    r1 = T → let (r2,σ2) = eval⟦f:x⟧σ1η in (r2,σ2[r2/t])
                    ; r1 = F → let (r2,σ2) = eval⟦g:x⟧σ1η in (r2,σ2[r2/t])
                    ; (⊥,σ1[⊥/t])

    αf:x ⇒ (r,σ[r/t])
          where r = cases x is
                      "an atom" ⇒ ⊥
                      φ ⇒ φ
                      <O1, . . . , On> ⇒ "f is predefined"
                                         → <(Fn f)O1, . . . , (Fn f)On>
                                         ; [f∘1st, . . . , f∘nth]:x

    /f:x ⇒ cases x is
             "an atom" ⇒ (⊥,σ[⊥/t])
             φ ⇒ (⊥,σ[⊥/t])
             <O1> ⇒ (O1,σ[O1/t])
             <O1, . . . , On> ⇒
                  predefined(f)
                      → let (r2,σ2) = eval⟦/f:<O2, . . . , On>⟧σ1η in
                              let (r3,σ3) = eval⟦f:<O1,r2>⟧σ2η in
                                    (r3,σ3[r3/t])
                  ; let (r2,σ2) = eval⟦f∘[1st,/f∘[2nd, . . . , nth]]:x⟧ σ1η
                                  in (r2,σ2[r2/t])

    []:x ⇒ (φ,σ[φ/t])

    [f1, . . . , fn]:x ⇒ let t' = ◂f1:x, . . . , fn:x▸ in (t',σ[t'/t])
```

$\bar{a}:x \Rightarrow (a,\sigma[a/t])$

$(bu\ f\ a):x \Rightarrow \text{let } (r_1,\sigma_1) = eval[\![f:\langle a,x\rangle]\!]\sigma\eta \text{ in } (r_1,\sigma_1[r_1/t])$

$$(while\ p\ f):x \Rightarrow \text{let } (r_1,\sigma_1) = eval[\![p:x]\!]\sigma\eta \text{ in}$$
$$r_1 = T \to \text{let } (r_2,\sigma_2) = eval[\![(while\ p\ f)\circ f:x]\!]\sigma_1\eta \text{ in } (r_2,\sigma_2[r_2/t])$$
$$; r_1 = F \to (x,\sigma_1[x/t])$$
$$; (\perp,\sigma_1[\perp/t])$$

$s^{th}:x \Rightarrow (r,\sigma[r/t])$ where r = cases x is

"an atom" $\Rightarrow \perp$

$\phi \Rightarrow \perp$

$\langle O_1,\ldots,O_n\rangle \Rightarrow s \le n \to O_s ; \perp$

$rs^{th}:x \Rightarrow (r,\sigma[r/t])$ where r = cases x is

"an atom" $\Rightarrow \perp$

$\phi \Rightarrow \perp$

$\langle O_1,\ldots,O_n\rangle \Rightarrow s \le n \to O_{n-s+1}$;$\perp$

$force \circ f:x \Rightarrow \text{let } (r_1,\sigma_1) = eval[\![f:x]\!]\sigma\eta \text{ in}$
$\text{let } (r_2,\sigma_2) = FORCE(r_1)\sigma_1\eta \text{ in } (r_2,\sigma_2[r_2/t])$

$apndl \circ f:x \Rightarrow (t,\sigma)$

$apndr \circ f:x \Rightarrow (t,\sigma)$

$[] \circ f:x \Rightarrow (\phi,\sigma[\phi/t])$

$[f_1,\ldots,f_n] \circ g:x \Rightarrow \text{let } t' = [f_1 \circ g:x,\ldots,f_n \circ g:x] \text{ in } (t',\sigma[t'/t])$

$\bar{a} \circ f:x \Rightarrow (a,\sigma[a/t])$

$atom \circ f:x \Rightarrow \text{let } (r_1,\sigma_1) = eval\ [f:x]\sigma\eta \text{ in } (r_2,\sigma_1[r_2/t])$
where r_2 = cases r_1 is

"an atom" $\Rightarrow T$

"a sequence" $\Rightarrow F$

$[g_1:y,\ldots,g_n:y] \Rightarrow F$

$apndl \circ g:y \Rightarrow F$

$apndr \circ g:y \Rightarrow F$

$$\text{distl} \circ \text{f:x} \Rightarrow \text{let } (r_1,\sigma_1) = \text{eval}[\![(\text{null} \circ 2^{\text{nd}} \rightarrow \bar{\phi} ; \text{apndl} \circ [[1^{\text{st}}, 1^{\text{st}} \circ 2^{\text{nd}}], \text{distl} \circ [1^{\text{st}}, \text{tl} \circ 2^{\text{nd}}]]) \circ \text{f:x}]\!]\sigma\eta \text{ in } (r_1, \sigma_1[r_1/\text{t}])$$

$$\text{distr} \circ \text{f:x} \Rightarrow \text{let } (r_1,\sigma_1) = \text{eval}[\![(\text{null} \circ 1^{\text{st}} \rightarrow \bar{\phi} ; \text{apndl} \circ [[1^{\text{st}} \circ 1^{\text{st}}, 2^{\text{nd}}], \text{distr} \circ [\text{tl} \circ 1^{\text{st}}, 2^{\text{nd}}]]) \circ \text{f:x}]\!]\sigma\eta \text{ in } (r_1, \sigma_1[r_1/\text{t}])$$

$$\text{id} \circ \text{f:x} \Rightarrow \text{let } (r_1,\sigma_1) = \text{eval}[\![\text{f:x}]\!]\sigma\eta \text{ in } (r_1, \sigma_1[r_1/\text{t}])$$

$$\text{length} \circ \text{f:x} \Rightarrow \text{let } (r_1,\sigma_1) = \text{eval}[\![\text{f:x}]\!]\sigma\eta \text{ in}$$

cases r_1 is

- "an atom" $\Rightarrow (\perp, \sigma_1[\perp/\text{t}])$
- $\phi \Rightarrow (0, \sigma_1[0/\text{t}])$
- $\langle O_1, \ldots, O_n \rangle \Rightarrow (n, \sigma_1[n/\text{t}])$
- $[g_1\text{:y}, \ldots, g_n\text{:y}] \Rightarrow (n, \sigma_1[n/\text{t}])$
- $\text{apndl} \circ \text{g:y} \Rightarrow$ let $(r_2,\sigma_2) = \text{eval}[\![\text{length} \circ 2^{\text{nd}} \circ \text{g:y}]\!]\sigma_1\eta$ in $(r_3, \sigma_2[r_3/\text{t}])$ where $r_3 = r_2 + 1$
- $\text{apndr} \circ \text{g:y} \Rightarrow$ let $(r_2,\sigma_2) = \text{eval}[\![\text{length} \circ 1^{\text{st}} \circ \text{g:y}]\!]\sigma_1\eta$ in $(r_3, \sigma_2[r_3/\text{t}])$ where $r_3 = r_2 + 1$

$$\text{null} \circ \text{f:x} \Rightarrow \text{let } (r_1,\sigma_1) = \text{eval}[\![\text{f:x}]\!]\sigma\eta \text{ in } (r_2, \sigma_1[r_2/\text{t}])$$

where r_2 = cases r_1 is

- "an atom" $\Rightarrow$ F
- $\phi \Rightarrow$ T
- $\langle O_1, \ldots, O_n \rangle \Rightarrow$ F
- $[g_1\text{:y}, \ldots, g_n\text{:y}] \Rightarrow$ F
- $\text{apndl} \circ \text{g:y} \Rightarrow$ F
- $\text{apndr} \circ \text{g:y} \Rightarrow$ F

$$
\begin{aligned}
&\text{reverse} \circ \text{f:x} \Rightarrow \text{let } (r_1,\sigma_1) = \text{eval}[\![\text{f:x}]\!]\sigma\eta \text{ in } (r_2,\sigma_1[r_2/\,t]) \\
&\qquad \text{where } r_2 = \text{cases } r_1 \text{ is} \\
&\qquad\qquad \text{"an atom"} \Rightarrow \bot \\
&\qquad\qquad \phi \Rightarrow \phi \\
&\qquad\qquad \langle O_1, \ldots, O_n\rangle \Rightarrow \langle O_n, \ldots, O_1\rangle \\
&\qquad\qquad [g_1\text{:y}, \ldots, g_n\text{:y}] \Rightarrow [g_n\text{:y}, \ldots, g_1\text{:y}] \\
&\qquad\qquad \text{apndl} \circ \text{g:y} \Rightarrow \text{apndr} \circ [\text{reverse} \circ 2^{\text{nd}}, 1^{\text{st}}] \circ \text{g:y} \\
&\qquad\qquad \text{apndr} \circ \text{g:y} \Rightarrow \text{apndl} \circ [\text{r1}^{\text{st}}, \text{reverse} \circ 1^{\text{st}}] \circ \text{g:y}
\end{aligned}
$$

$$
\begin{aligned}
&\text{rotl} \circ \text{f:x} \Rightarrow \\
&\quad \text{let } (r_1,\sigma_1) = \text{eval}[\![(\text{null} \rightarrow \vec{\phi}\ ;\ \text{apndr} \circ [\text{tl}, 1^{\text{st}}]) \circ \text{f:x}]\!]\sigma\eta \text{ in} \\
&\qquad\qquad (r_1,\sigma_1[r_1/\,t])
\end{aligned}
$$

$$
\begin{aligned}
&\text{rotr} \circ \text{f:x} \Rightarrow \\
&\quad \text{let } (r_1,\sigma_1) = \text{eval}[\![(\text{null} \rightarrow \vec{\phi}\ ;\ \text{apndl} \circ [\text{r1}^{\text{st}}, \text{tlr}]) \circ \text{f:x}]\!]\sigma\eta \text{ in} \\
&\qquad\qquad (r_1,\sigma_1[r_1/\,t])
\end{aligned}
$$

$$
\begin{aligned}
&\text{tl} \circ \text{f:x} \Rightarrow \text{let } (r_1,\sigma_1) = \text{eval}[\![\text{f:x}]\!]\sigma\eta \text{ in} \\
&\qquad\qquad \text{cases } r_1 \text{ is} \\
&\qquad\qquad \text{"an atom"} \Rightarrow (\bot, \sigma_1[\bot/\,t]) \\
&\qquad\qquad \phi \Rightarrow (\bot, \sigma_1[\bot/\,t]) \\
&\qquad\qquad \langle O_1, \ldots, O_n\rangle \Rightarrow (r_2,\sigma_1[r_2/\,t]) \\
&\qquad\qquad\quad \text{where } r_2 = \langle O_2, \ldots, O_n\rangle \\
&\qquad\qquad [g_1\text{:y}, \ldots, g_n\text{:y}] \Rightarrow (r_2,\sigma_1[r_2/\,t]) \\
&\qquad\qquad\quad \text{where } r_2 = [g_2\text{:y}, \ldots, g_n\text{:y}] \\
&\qquad\qquad \text{apndl} \circ \text{g:y} \Rightarrow \text{let } (r_2,\sigma_2) = \text{eval}[\![2^{\text{nd}} \circ \text{g:y}]\!]\sigma_1\eta \text{ in} \\
&\qquad\qquad\qquad (r_2,\sigma_2[r_2/\,t]) \\
&\qquad\qquad \text{apndr} \circ \text{g:y} \Rightarrow \\
&\qquad\qquad\quad \text{let } (r_2,\sigma_2) = \text{eval}[\![(\text{null} \circ 1^{\text{st}} \rightarrow \vec{\phi} \\
&\qquad\qquad\qquad\qquad ;\ \text{apndr} \circ [\text{tl} \circ 1^{\text{st}}, 2^{\text{nd}}]) \circ \text{g:y}]\!]\sigma_1\eta \\
&\qquad\qquad\qquad\qquad \text{in } (r_2,\sigma_2[r_2/\,t])
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{tlr} \circ \mathrm{f{:}x} \Rightarrow \text{let } (r_1,\sigma_1) = \mathrm{eval}[\![\mathrm{f{:}x}]\!]\sigma\eta \text{ in} \\
&\qquad \text{cases } r_1 \text{ is} \\
&\qquad \text{"an atom"} \Rightarrow (\bot,\sigma_1[\bot/\mathrm{t}]) \\
&\qquad \phi \Rightarrow (\bot,\sigma_1[\bot/\mathrm{t}]) \\
&\qquad \langle O_1,\ldots,O_n\rangle \Rightarrow (r_2,\sigma_1[r_2/\mathrm{t}]) \quad \text{where } r_2 = \langle O_1,\ldots,O_{n-1}\rangle \\
&\qquad [g_1{:}y,\ldots,g_n{:}y] \Rightarrow (r_2,\sigma_1[r_2/\mathrm{t}]) \\
&\qquad\quad \text{where } r_2 = [g_1{:}y,\ldots,g_{n-1}{:}y] \\
&\qquad \mathrm{apndl} \circ \mathrm{g{:}y} \Rightarrow \text{let } (r_2,\sigma_2) = \\
&\qquad\qquad \mathrm{eval}[\![(\mathrm{null} \circ 2^{\mathrm{nd}} \rightarrow \bar{\phi} \\
&\qquad\qquad\qquad ; \mathrm{apndl} \circ [1^{\mathrm{st}},\mathrm{tlr} \circ 2^{\mathrm{nd}}]) \circ \mathrm{g{:}y}]\!] \\
&\qquad\qquad \text{in } (r_2,\sigma_2[r_2/\mathrm{t}]) \\
&\qquad \mathrm{apndr} \circ \mathrm{g{:}y} \Rightarrow \text{let } (r_2,\sigma_2) = \mathrm{eval}[\![1^{\mathrm{st}} \circ \mathrm{g{:}y}]\!]\sigma_1\eta \text{ in } (r_2,\sigma_2[r_2/\mathrm{t}])
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{trans} \circ \mathrm{f{:}x} \Rightarrow \text{let } (r_1,\sigma_1) = \mathrm{eval}[\![(/\mathrm{and} \circ \alpha\mathrm{null} \rightarrow \bar{\phi} \\
&\qquad\qquad\qquad\qquad ; \mathrm{apndl} \circ [\alpha 1^{\mathrm{st}},\mathrm{trans} \circ \alpha\mathrm{tl}]) \circ \mathrm{f{:}x}]\!]\,\sigma\eta \\
&\qquad\qquad \text{in } (r_1,\sigma_1[r_1/\mathrm{t}])
\end{aligned}
$$

$$
\begin{aligned}
&1^{\mathrm{st}} \circ \mathrm{f{:}x} \Rightarrow \text{let } (r_1,\sigma_1) = \mathrm{eval}[\![\mathrm{f{:}x}]\!]\sigma\eta \text{ in} \\
&\qquad\qquad \text{cases } r_1 \text{ is} \\
&\qquad\qquad \text{"an atom"} \Rightarrow (\bot,\sigma_1[\bot/\mathrm{t}]) \\
&\qquad\qquad \phi \Rightarrow (\bot,\sigma_1[\bot/\mathrm{t}]) \\
&\qquad\qquad \langle O_1,\ldots,O_n\rangle \Rightarrow (O_1,\sigma_1[O_1/\mathrm{t}]) \\
&\qquad\qquad [g_1{:}y,\ldots,g_n{:}y] \Rightarrow \text{let } (r_2,\sigma_2) = \mathrm{eval}[\![g_1{:}y]\!]\sigma_1\eta \text{ in} \\
&\qquad\qquad\qquad\qquad (r_2,\sigma_2[r_2/\mathrm{t}]) \\
&\qquad\qquad \mathrm{apndl} \circ \mathrm{g{:}y} \Rightarrow \text{let } (r_2,\sigma_2) = \mathrm{eval}[\![1^{\mathrm{st}} \circ \mathrm{g{:}y}]\!]\,\sigma_1\eta \text{ in} \\
&\qquad\qquad\qquad\qquad (r_2,\sigma_2[r_2/\mathrm{t}]) \\
&\qquad\qquad \mathrm{apndr} \circ \mathrm{g{:}y} \Rightarrow \text{let}(r_2,\sigma_2) = \\
&\qquad\qquad\qquad \mathrm{eval}[\![(\mathrm{null} \circ 1^{\mathrm{st}} \rightarrow 2^{\mathrm{nd}} \\
&\qquad\qquad\qquad\qquad ; 1^{\mathrm{st}} \circ 1^{\mathrm{st}}) \circ \mathrm{g{:}y}]\!]\sigma_1\eta \\
&\qquad\qquad\qquad\qquad \text{in } (r_2,\sigma_2[r_2/\mathrm{t}])
\end{aligned}
$$

$$
s^{\mathrm{th}} \circ \mathrm{f} \Rightarrow \text{let } (r_1,\sigma_1) = \mathrm{eval}[\![(s-1)^{\mathrm{th}} \circ \mathrm{tl} \circ \mathrm{f{:}x}]\!]\sigma\eta \text{ in } (r_1,\sigma_1[r_1/\mathrm{t}])
$$

$r1^{st} \circ f{:}x \Rightarrow$ let $(r_1,\sigma_1) = eval[\![f{:}x]\!]\sigma\eta$ in
cases r_1 is
"an atom" $\Rightarrow (\perp,\sigma_1[\perp/t])$
$\phi \Rightarrow (\perp,\sigma_1[\perp/t])$
$\langle O_1, \ldots, O_n\rangle \Rightarrow (O_n,\sigma_1[O_n/t])$
$[g_1{:}y, \ldots, g_n{:}y] \Rightarrow$
let $(r_2,\sigma_2) =$ $eval[\![g_n{:}y]\!]\sigma_1\eta$ in $(r_2,\sigma_2[r_2/t])$
$apndl \circ g{:}y \Rightarrow$
let $(r_2,\sigma_2) =$ $eval[\![(null \circ 2^{nd} \rightarrow 1^{st}$.
$; r1^{st} \circ 2^{nd}) \circ g{:}y]\!]\sigma_1\eta$ in
$(r_2,\sigma_2[r_2/t])$
$apndr \circ g{:}y \Rightarrow$
let$(r_2,\sigma_2) =$ $eval[\![2^{nd} \circ g{:}y]\!]\sigma_1\eta$ in $(r_2,\sigma_2[r_2/t])$

$rs^{th} \circ f{:}x \Rightarrow$ let $(r_1,\sigma_1) =$ $eval[\![r(s-1)^{th} \circ tlr \circ f{:}x]\!]\sigma\eta$ in
$(r_1,\sigma_1[r_1/t])$

$/f \circ g{:}x \Rightarrow$ let $(r_1,\sigma_1) = eval[\![g{:}x]\!]\sigma\eta$ in
cases r_1 is
"an atom" $\Rightarrow (\perp,\sigma_1[\perp/t])$
$\phi \Rightarrow (\perp,\sigma_1[\perp/t])$
$\langle O_1\rangle \Rightarrow (O_1,\sigma_1[O_1/t])$
$\langle O_1, \ldots, O_n\rangle \Rightarrow$
predefined(f) $\rightarrow$ let $(r_2,\sigma_2) =$ $eval[\![/f{:}\langle O_2, \ldots, O_n\rangle]\!]\sigma_1\eta$ in
let $(r_3,\sigma_3) =$
$eval[\![f{:}\langle O_1,r_2\rangle]\!]\sigma_2\eta$ in
$(r_3,\sigma_3[r_3/t])$
; let $(r_2,\sigma_2) =$
$eval[\![f \circ [1^{st},/f \circ [2^{nd}, \ldots, n^{th}]]{:}r_1]\!]\ \sigma_1\eta$
in $(r_2,\sigma_2[r_2/t])$
$[h{:}y] \Rightarrow$ let $(r_2,\sigma_2) = eval[\![h{:}y]\!]\sigma_1\eta$ in $(r_2,\sigma_2[r_2/t])$
$[h_1{:}y, \ldots, h_n{:}y] \Rightarrow$
let $(r_2,\sigma_2) =$ $eval[\![f \circ [h_1{:}y,/f \circ [h_2{:}y, \ldots, h_n{:}y]]]\!]\sigma_1\eta$ in
$(r_2,\sigma_2[r_2/t])$
$apndl \circ h{:}y \Rightarrow$
let $(r_2,\sigma_2) =$
$eval[\![(null \circ 2^{nd} \rightarrow 1^{st}\ ; f \circ [1^{st},/f \circ 2^{nd}]) \circ h{:}y]\!]\ \sigma_1\eta$
in $(r_2,\sigma_2[r_2/t])$
$apndr \circ h{:}y \Rightarrow$
let$(r_2,\sigma_2) =$
$eval[\![(null \circ tl \rightarrow 1^{st}\ ; f \circ [1^{st},/f \circ tl]) \circ apndr \circ h{:}y]\!]\sigma_1\eta$
in $(r_2,\sigma_2[r_2/t])$

$(\alpha f)\circ g{:}x \Rightarrow$
  let (r_1,σ_1) = eval$[\![g{:}x]\!]\sigma\eta$ in $(r_2,\sigma_1[r_2/t])$
    where r_2 = cases r_1 is
      "an atom" $\Rightarrow \perp$
      $\phi \Rightarrow \phi$
      $\langle O_1,\ldots,O_n\rangle \Rightarrow$ predefined(f)
        $\rightarrow \langle(\text{Fn f})O_1,\ldots,(\text{Fn f})O_n\rangle$
        ; $[f\circ 1^{st},\ldots,f\circ n^{th}]{:}r_1$
      $[h_1{:}y,\ldots,h_n{:}y] \Rightarrow [f\circ h_1{:}y,\ldots,f\circ h_n{:}y]$
      apndl$\circ$h:y $\Rightarrow$ apndl$\circ[f\circ 1^{st},\alpha f\circ 2^{nd}]\circ$h:y
      apndr$\circ$h:y $\Rightarrow$ apndr$\circ[\alpha f\circ 1^{st},f\circ 2^{nd}]\circ$h:y

(bu f a)$\circ$g:x $\Rightarrow$ let (r_1,σ_1) = eval$[\![f\circ[\bar{a},g]{:}x\sigma\eta$ in $(r_1,\sigma_1[r_1/t])$

(while p f)$\circ$g:x $\Rightarrow$
  let (r_1,σ_1) = eval$[\![p\circ g{:}x]\!]\sigma\eta$ in
    r_1 = T $\rightarrow$ let (r_2,σ_2) = eval$[\![$(while p f)$\circ f\circ g{:}x]\!]\sigma_1\eta$ in
      $(r_2,\sigma_2[r_2/t])$
    ; r_1 = F $\rightarrow$ let (r_2,σ_2) = eval$[\![g{:}x]\!]\sigma_1\eta$ in $(r_2,\sigma_2[r_2/t])$
    ; $(\perp,\sigma_1[\perp/t])$

(p $\rightarrow$ f ; g)$\circ$h:x $\Rightarrow$
  let (r_1,σ_1) = eval$[\![p\circ h{:}x]\!]\sigma\eta$ in
    r_1 = T $\rightarrow$ let (r_2,σ_2) = eval$[\![f\circ h{:}x]\!]\sigma_1\eta$ in $(r_2,\sigma_2[r_2/t])$
    ; r_1 = F $\rightarrow$ let (r_2,σ_2) = eval$[\![g\circ h{:}x]\!]\sigma_1\eta$ in $(r_2,\sigma_2[r_2/t])$
    ; $(\perp,\sigma_1[\perp/t])$

f$\circ$g:x $\Rightarrow$ let (r_1,σ_1) = eval$[\![g{:}x]\!]\sigma\eta$ in
    let (r_2,σ_2) = FORCE$(r_1)\sigma_1\eta$ in
      $(r_3,\sigma_2[r_3/t])$ where $r_3 = (\text{Fn f})r_2$
  where f$\in$ {add, and, div, eq, ge, gt, le, lt, mul, ne, not, or, sub}

```
FORCE(t)ση ≡ cases t is
      an object ⇒ (t,s)
      [f1:x, . . . , fn:x] ⇒
         let (l,σ1) = <FORCE(f1:x)ση, . . . , FORCE (fn:x)ση>
             in (<l>,σ1[<l>/ t])
      apndl ∘ g:x ⇒
         let (r1,σ1) = eval⟦g:x⟧ση in
            "r1 is an object" → (r2,σ1[r2/ t])   where r2 = APNDL(r1)
                  ; let (r2,σ2) = FORCE(r1)σ1η in
                     (r3,σ2[r3/ t])   where r3 = APNDL(r2)
      apndr ∘ g:x ⇒
         let (r1,σ1) = eval⟦g:x⟧ση in
            "r1 is an object" → (r2,σ1[r2/ t])   where r2 = APNDR(r1)
               ; let (r2,σ2) = FORCE(r1)σ1η in
                  (r3,σ2[r3/ t])   where r3 = APNDR(r2)
```

APPENDIX B

SEMANTIC DEFINITION OF FP

The Domains of Meaning

s-env = identifier → s-function
s-function = s-object → s-object
s-object = s-atom + s-sequence + ⊥
s-sequence = s-object*
s-atom = integer + literal + boolean

(Where identifier is the primitive domain of identifiers, integer is the primitive domain of integers, literal is the primitive domain of characters and character strings, boolean is {true,false}, and ⊥ is "bottom" or "undefined".)

The Functions of Meaning

Meaning : (object + f-expr + application + definition)* →
(s-function + s-object)* × s-env

Meaning' : (object + f-expr + application + definition)* → s-env →
(s-function + s-object)* × s-env

M : (object + application + definition + f-expr) → s-env →
(s-function + s-object) × s-env

F : f–expr → s–env → (s–object → s–object) + ⊥

---

notation: a.l means "item a appended onto list l"
e{e'} means "environment e updated by environment e'"

$\text{Meaning}[\![(t_1, \ldots, t_n)]\!] \equiv \text{Meaning}'[\![(t_1, \ldots, t_n)]\!]\phi$
where ϕ is an empty s-env

$\text{Meaning}'[\![(t_1, \ldots, t_n)]\!]\eta_\sigma \equiv$ **let** $(r_\sigma, \eta_{1_\sigma}) = \in [\![t_1]\!]\eta_\sigma$ in
let $(L_\sigma, \eta_{2_\sigma}) = \text{Meaning}'[\![(t_2, \ldots, t_n]\!]\eta_{1_\sigma}$ in
$n = 0 \rightarrow ((), \eta_\sigma)\ ;\ (r_\sigma . L_\sigma, \eta_{2_\sigma})$

$M[\![0]\!]\eta_\sigma \equiv <0_\sigma, \eta_\sigma>$ (0_σ is the semantic object represented by AST 0)

$M[\![f]\!]\eta_\sigma \equiv <F[\![f]\!]\eta_\sigma, \eta_\sigma>$ (where f is an f-expr)

$M[\![\text{def i t}]\!]\eta_\sigma \equiv F[\![t]\!]\eta_\sigma = \phi$
$\rightarrow <i, \eta_\sigma\{Y.F[\![t]\!]\eta_\sigma / i\}>$
$; <\perp, \eta_\sigma>$

$M[\![f{:}x]\!]\eta_\sigma \equiv <M[\![f]\!]\eta_\sigma : M[\![x]\!]\eta_\sigma, \eta_\sigma>$

$F[\![s^{th}]\!]\eta_\sigma \equiv \lambda 0.\ 0 = <0_1, \ldots, 0_n> \rightarrow n> = s \rightarrow 0_s\ ;\ \perp$
$;\perp$

$F[\![t]\!]\eta_\sigma \equiv \lambda 0.\ 0 = <0_1> \rightarrow <>$
$;0 = <0_1, \ldots, 0_n> \ \&\ n> = 2 \rightarrow <0_2, \ldots, 0_n>$
$,\perp$

$F[\![id]\!]\eta_\sigma \equiv \lambda 0.0$

$F[\![atom]\!]\eta_\sigma \equiv \lambda 0.\ 0$ "is an atom" $\rightarrow$ true
$;0 \neq \perp \rightarrow$ false
$;\perp$

$F[\![eq]\!]\eta_\sigma \equiv \lambda 0.\ 0 = <0_1, 0_2>$
$\rightarrow 0_1 = 0_2 \rightarrow$ true
;false

$F[\![null]\!]\eta_\sigma \equiv \lambda 0.\ 0 = <> \rightarrow$ true
$;0 \neq \perp \rightarrow$ false
$;\perp$

$F[\![reverse]\!]\eta_\sigma \equiv \lambda 0.\ 0 = <> \rightarrow <>$
$;0 = <0_1, \ldots, 0_n> \rightarrow <0_n, \ldots, 0_1>$
$;\perp$

$$F[\![\text{distl}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, \langle\rangle\rangle \rightarrow \langle\rangle$$
$$;0 = \langle O_1, \langle O_2, \ldots, O_n\rangle\rangle \rightarrow \langle\langle O_1, O_2\rangle, \ldots, \langle O_1, O_n\rangle\rangle$$
$$;\perp$$

$$F[\![\text{distr}]\!]\eta_\sigma \equiv \lambda 0.\ 0 = \langle\langle\rangle, O_1\rangle \rightarrow \langle\rangle$$
$$;0 = \langle\langle O_2, \ldots, O_n\rangle, O_1\rangle \rightarrow \langle\langle O_2, O_1\rangle, \ldots, \langle O_n, O_1\rangle\rangle$$
$$;\perp$$

$$F[\![\text{length}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle\rangle \rightarrow 0$$
$$;0 = \langle O_1, \ldots, O_n\rangle \rightarrow n$$
$$;\perp$$

$$F[\![\text{add}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 + O_2$$
$$;\perp$$

$$F[\![\text{sub}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 - O_2$$
$$;\perp$$

$$F[\![\text{mul}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 * O_2$$
$$;\perp$$

$$F[\![\text{div}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 / O_2$$
$$;\perp$$

$$F[\![\text{gt}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 > O_2$$
$$;\perp$$

$$F[\![\text{ge}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 \geq O_2$$
$$;\perp$$

$$F[\![\text{lt}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 < O_2$$
$$;\perp$$

$$F[\![\text{le}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 \leq O_2$$
$$;\perp$$

$$F[\![\text{ne}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle O_1, O_2\rangle \ \&\ O_1, O_2 \text{ are numbers} \rightarrow O_1 \neq O_2$$
$$;\perp$$

$$F[\![\text{trans}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle\langle\rangle, \ldots, \langle\rangle\rangle \rightarrow \langle\rangle$$
$$;0 = \langle\langle O_{11}, \ldots, O_{1m}\rangle, \ldots, \langle O_{n1}, \ldots, O_{nm}\rangle\rangle$$
$$\rightarrow \langle\langle O_{11}, \ldots, O_{n1}\rangle, \ldots, \langle O_{1m}, \ldots, O_{nm}\rangle\rangle$$
$$;\perp$$

$$F[\![\text{and}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle \text{true,true} \rangle \rightarrow \text{true}$$
$$;0 = \langle \text{true,false} \rangle \text{ or } 0 = \langle \text{false,true} \rangle \text{ or } 0 = \langle \text{false,false} \rangle \rightarrow \text{false}$$
$$;\perp$$

$$F[\![\text{or}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle \text{true,true} \rangle \text{ or } 0 = \langle \text{true,false} \rangle \text{ or } 0 = \langle \text{false,true} \rangle \rightarrow \text{true}$$
$$;0 = \langle \text{false,false} \rangle \rightarrow \text{false}$$
$$;\perp$$

$$F[\![\text{not}]\!]\eta_\sigma \equiv \lambda 0.0 = \text{true} \rightarrow \text{false}$$
$$;0 = \text{false} \rightarrow \text{true}$$
$$;\perp$$

$$F[\![\text{apndl}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle 0_1, \langle\rangle \rangle \rightarrow \langle 0_1 \rangle$$
$$;0 = \langle 0_1, \langle 0_2, \ldots, 0_n \rangle \rangle \rightarrow \langle 0_1, 0_2, \ldots, 0_n \rangle$$
$$;\perp$$

$$F[\![\text{apndr}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle \langle\rangle, 0_1 \rangle \rightarrow \langle 0_1 \rangle$$
$$;0 = \langle \langle 0_2, \ldots, 0_n \rangle, 0_1 \rangle \rightarrow \langle 0_2, \ldots, 0_n, 0_1 \rangle$$
$$;\perp$$

$$F[\![s^{th}r]\!]\eta_\sigma \equiv \lambda 0.0 = \langle 0_1, \ldots, 0_n \rangle \;\&\; n \geq s \rightarrow 0_{n-s+1}$$
$$;\perp$$

$$F[\![\text{tlr}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle 0_1 \rangle \rightarrow \langle\rangle$$
$$;0 = \langle 0_1, \ldots, 0_n \rangle \;\&\; n \geq 2 \rightarrow \langle 0_1, \ldots, 0_{n-1} \rangle$$
$$;\perp$$

$$F[\![\text{rotl}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle\rangle \rightarrow \langle\rangle$$
$$;0 = \langle 0_1 \rangle \rightarrow \langle 0_1 \rangle$$
$$;0 = \langle 0_1, \ldots, 0_n \rangle \;\&\; n \geq 2 \rightarrow \langle 0_2, \ldots, 0_n, 0_1 \rangle$$
$$;\perp$$

$$F[\![\text{rotr}]\!]\eta_\sigma \equiv \lambda 0.0 = \langle\rangle \rightarrow \langle\rangle$$
$$;0 = \langle 0_1 \rangle \rightarrow \langle 0_1 \rangle$$
$$;0 = \langle 0_1, \ldots, 0_n \rangle \;\&\; n \geq 2 \rightarrow \langle 0_n, 0_1, \ldots, 0_{n-1} \rangle$$
$$;\perp$$

$$F[\![f \circ g]\!]\eta_\sigma \equiv \lambda 0.F[\![f]\!]\eta_\sigma : (F[\![g]\!]\eta_\sigma : 0)$$

$$F[\![[f_1, \ldots, f_n]]\!]\eta_\sigma \equiv \lambda 0.\langle F[\![f_1]\!]\eta_\sigma : 0, \ldots, F[\![f_n]\!]\eta_\sigma : 0 \rangle$$

$$F[\![p \to f;g]\!]\eta_\sigma \equiv \lambda O.F[\![p]\!]\eta_\sigma:O = true \to F[\![f]\!]\eta_\sigma:O$$
$$;F[\![p]\!]\eta_\sigma:O = false \to F[\![g]\!]\eta_\sigma:O$$
$$;\perp$$

$$F[\![\bar{a}]\!]\eta_\sigma \equiv \lambda O.a$$

$$F[\![/\, f]\!]\eta_\sigma \equiv \lambda O.O = \langle O_1\rangle \to O_1$$
$$;O = \langle O_1,\ldots,O_n\rangle \;\&\; n > = 2$$
$$\to F[\![f]\!]\eta_\sigma:\langle O_1, F[\![/\, f]\!]\eta_\sigma:\langle O_2,\ldots,O_n\rangle\rangle$$
$$;\perp$$

$$F[\![\alpha f]\!]\eta_\sigma \equiv \lambda O.O = \langle\rangle \to \langle\rangle$$
$$;O = \langle O_1, \ldots, O_n\rangle$$
$$\to \langle F[\![|gf]\!]\eta_\sigma:O_1, \ldots, F[\![f]\!]:O_n\rangle$$
$$;\perp$$

$$F[\![bu\ f\ a]\!]\eta_\sigma \equiv \lambda O.\ F[\![f]\!]\eta_\sigma:\langle a,O\rangle$$

$$F[\![while\ p\ f]\!]\eta_\sigma \equiv Y\lambda w.\lambda O.F[\![p]\!]\eta_\sigma:O = true$$
$$\to w:(F[\![f]\!]\eta_\sigma:O)$$
$$;F[\![p]\!]\eta_\sigma:O = false \to O$$
$$;\perp$$

$$F[\![i]\!]\eta_\sigma \equiv \lambda O.\eta_\sigma:i \neq \phi \to F[\![\eta_\sigma:i]\!]\eta_\sigma:O$$
$$;\perp$$

APPENDIX C

CORRECTNESS OF TRANSFORMATIONS

For each transformation eval⟦e⟧ = eval⟦e'⟧ performed by the interpreter, we will show that e is less defined than or equal to e', i.e. $M(e) \sqsubseteq M(e')$. The function

$$M : (\text{object}+\text{application}+\text{definition}+\text{f-expr}) \to \text{s-env} \to (\text{s-function}+\text{s-object}) \times \text{s-env}$$

is a semantic function defined in Appendix B.

The following liberties will be taken:

1) Objects will denote their meanings. For example: $M(\perp) = \perp$, $M(<1,2>) = <1,2>$.

2) The environment and state arguments to *eval* and *M* have no effect on the proofs, and will therefore be omitted.

3) If eval⟦e ∘ f⟧ = eval⟦e' ∘ f⟧, then by the theorem on p. 26 it will be sufficient to show that $M(e) \sqsubseteq M(e')$.

4) Obvious steps are left out of the proofs. For example:

$$M(\text{tl}) : M(<a_1, \ldots, a_n>) = (\lambda <O_1, \ldots, O_n> . <O_2, \ldots, O_n>) : <a_1, \ldots, a_n> = <a_2, \ldots, a_n>$$

will be reduced to a single step.

5) If the proofs of two transformations are very similar, then only one of the proofs is given.

6) For the transformation eval⟦f ∘ g⟧ = eval⟦h⟧, we want to show that if $M(f \circ g:x) \neq \perp$ then $M(f \circ g:x) = M(h:x)$. The set of all x s.t. $M(f \circ g:x) \neq \perp$ is a subset of the set of all x s.t. $M(g:x) \neq \perp$. For most proofs we will show that $M(f \circ g:x) = M(h:x)$ wherever $M(g:x) \neq \perp$, which will clearly cover all cases where $M(f \circ g:x) \neq \perp$.

7) Some of the proofs will take advantage of the algebra of programs [1].

Theorem T1: $M(distl) \sqsubseteq M(null \circ 2^{nd} \rightarrow \bar{\phi}$
$; apndl \circ [[1^{st}, 1^{st} \circ 2^{nd}], distl \circ [1^{st}, tl \circ 2^{nd}]])$

Proof:

case 1: $x = \langle a, \langle\rangle\rangle$

$M(distl:\langle a,\langle\rangle\rangle) = \langle\rangle$ (M(distl))
$M((null \circ 2^{nd} \rightarrow \bar{\phi}\ ;\ldots):\langle a,\langle\rangle\rangle) = \phi$ (cond,constant)

case 2: $x = \langle a, \langle b_1, \ldots, b_n\rangle\rangle$

$M(distl:\langle a,\langle b_1, \ldots, b_n\rangle\rangle) = \langle\langle a,b_1\rangle,\ldots,\langle a,b_n\rangle\rangle$ (M(distl))
$M((null \circ 2^{nd} \rightarrow \bar{\phi}\ ;\ldots):\langle a,\langle b_1, \ldots, b_n\rangle\rangle)$
$= M(apndl \circ [[1^{st}, 1^{st} \circ 2^{nd}], distl \circ [1^{st}, tl \circ 2nd]]:\langle a,\langle b_1, \ldots, b_n\rangle\rangle$ (cond)
$= M(apndl:\langle\langle a,b_1\rangle, distl:\langle a,\langle b_2, \ldots, b_n\rangle\rangle\rangle)$ (constr,selectors)
$= M(apndl:\langle\langle a,b_1\rangle,\langle\langle a,b_2\rangle,\ldots,\langle a,b_n\rangle\rangle\rangle)$ (M(distl)
$= \langle\langle a,b_1\rangle,\ldots,\langle a,b_n\rangle\rangle$ (M(apndl))

∎

Theorem T2: $M(distr) \sqsubseteq M(null \circ 1^{st} \rightarrow \bar{\phi}$
$; apndr \circ [[1^{st} \circ 1^{st}, 2^{nd}], distr \circ [tl \circ 1^{st}, 2^{nd}]])$

Proof: similar to proof of T1

Theorem T3: $M(id \circ f) \sqsubseteq M(f)$

Proof: III.2 in [1]

Theorem T4: $M(reverse \circ apndl) \sqsubseteq M(apndr \circ [reverse \circ 2^{nd}, 1st])$

Proof:

case 1: $x = \langle a, \langle\rangle\rangle$

$M(reverse \circ apndl:\langle a,\langle\rangle\rangle)$
$= M(reverse):M(apndl:\langle a,\langle\rangle\rangle)$ (comp)
$= M(reverse):\langle a\rangle$ (M(apndl))
$= \langle a\rangle$ (M(reverse))
$M(apndr \circ [reverse \circ 2^{nd}, 1^{st}]:\langle a,\langle\rangle\rangle)$
$= M(apndr):M([reverse \circ 2^{nd}, 1^{st}]:\langle a,\langle\rangle\rangle)$ (comp)
$= M(apndr):M(\langle reverse:\langle\rangle, a\rangle)$ (constr)
$= M(apndr):\langle\langle\rangle, a\rangle$ (M(reverse))

$= \langle a \rangle$ (M(apndr))

case 2: $x = \langle a, \langle b_1, \ldots, b_n \rangle \rangle$

$M(\text{reverse} \circ \text{apndl}:\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$
$= M(\text{reverse}):M(\text{apndl}):\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (comp)
$= M(\text{reverse}):\langle a, b_1, \ldots, b_n \rangle$ (M(apndl))
$= \langle b_n, \ldots, b_1, a \rangle$ (M(reverse))
$M(\text{apndr} \circ [\text{reverse} \circ 2^{nd}, 1^{st}]:\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$
$= M(\text{apndr}):M([\text{reverse} \circ 2^{nd}, 1st]:\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (comp)
$= M(\text{apndr}):M(\langle \text{reverse}:\langle b_1, \ldots, b_n \rangle, a \rangle)$ (selectors,constr)
$= M(\text{apndr}):\langle\langle b_1, \ldots, b_n \rangle, a \rangle$ (M(reverse))
$= \langle b_1, \ldots, b_n, a \rangle$

■

Theorem T5: $M(\text{reverse} \circ \text{apndr}) \sqsubseteq M(\text{apndl} \circ [r1^{st}, \text{reverse} \circ 1^{st}])$

Proof: similar to proof of T4

Theorem T6: $M(\text{rotl}) \sqsubseteq M(\text{null} \rightarrow \bar{\phi}\ ;\ \text{apndr} \circ [\text{tl}, 1^{st}])$

Proof:

case 1: $x = \phi$

$M(\text{rotl}:\phi) = \phi$ (M(rotl))
$M((\text{null} \rightarrow \bar{\phi}\ ;\ \text{apndr} \circ [\text{tl}, 1^{st}]):\phi) = \phi$ (cond,M(null))

case 2: x = <a1,...,an>

$M(\text{rotl}:\langle a_1, \ldots, a_n \rangle) = \langle X6, a_1 \rangle$ (M(rotl))
$M((\text{null} \rightarrow \bar{\phi}; \text{apndr} \circ [\text{tl}, 1^{st}]):\langle a_1, \ldots, a_n \rangle)$
$= M(\text{apndr} \circ [\text{tl}, 1^{st}]:\langle a_1, \ldots, a_n \rangle)$ (cond)
$= M(\text{apndr}:\langle\langle a_2, \ldots, a_n \rangle, a_1 \rangle)$ (comp,tl,selectors)
$= \langle a_2, \ldots, a_n, a_1 \rangle$ (apndr)

■

Theorem T7: $M(\text{rotr}) \sqsubseteq M(\text{null} \rightarrow \bar{\phi}\ ;\ \text{apndl} \circ [r1^{st}, \text{tlr}])$

Proof: similar to proof of T6

Theorem T8: $M(\text{tl} \circ \text{apndr}) \sqsubseteq M(\text{null} \circ 1^{st} \rightarrow \bar{\phi}\ ;\ \text{apndr} \circ [\text{tl} \circ 1^{st}, 2^{nd}])$

Proof:

$M(tl \circ apndr:<\phi,a>)$
$= M(tl):M(apndr:<\phi,a>)$ (comp)
$= M(tl):<a>$ (apndr)
$= \phi$ (tl)
$M((null \circ 1^{st} \rightarrow \vec{\phi}\ ;\ ...):<\phi,a>) = \phi$ (cond,null,sel,constant)

case 2: $x = <<b_1, \ldots, b_n>,a>$

$M(tl \circ apndr:<<b_1, \ldots, b_n>,a>)$
$= M(tl):M(apndr:<<b_1, \ldots, b_n>,a)$ (comp)
$= M(tl):<b_1, \ldots, b_n,a>$ (apndr)
$= <b_2, \ldots, b_n,a>$ (tl)
$M((null \circ 1^{st}...):<<b_1, \ldots, b_n,a>)$
$= M(apndr \circ [tl \circ 1^{st},2^{nd}]:<<b_1, \ldots, b_n>,a>)$ (cond)
$= M(apndr:<<b_2, \ldots, b_n>,a>)$ (constr,selectors)
$= <b_2, \ldots, b_n,a>$

■

Theorem T9: $M(tlr \circ apndl) \sqsubseteq M(null \circ 2^{nd} \rightarrow \vec{\phi}\ ;\ apndl \circ [1^{st},tlr \circ 2^{nd}])$

Proof: similar to proof of T8

Theorem T10: $M(tlr \circ apndr) \sqsubseteq M(1^{st})$

Proof: similar to proof on page 29

Theorem T11: $M(trans) \sqsubseteq M(/and \circ \alpha null \rightarrow \vec{\phi}\ ;\ apndl \circ [\alpha 1^{st},trans \circ \alpha tl])$

Proof:

case 1: $x = <\phi, \ldots, \phi>$

$M(trans:<\phi, \ldots, \phi>) = \phi$ (trans)
$M(/and \circ \alpha null \rightarrow \vec{\phi}\ ;\ ...):<\phi, \ldots, \phi>) = \phi$ (cond)

case 2: $x = <<a_{1_1}, \ldots, a_{1_n}>,...,<a_{m_1}, \ldots, a_{m_n}>>$

$M(trans:<<a_{1_1}, \ldots, a_{1_n}>,...,<a_{m_1}, \ldots, a_{m_n}>>)$
$= <<a_{1_1}, \ldots, a_{m_1}>,...,<a_{1_n}, \ldots, a_{m_n}>>$ (trans)
$M(/and \circ \alpha null ...):<<a_{1_1}, \ldots, a_{1_n}>,...,<a_{m_1}, \ldots, a_{m_n}>>)$
$= M(apndl \circ [\alpha 1^{st},trans \circ \alpha tl]:<<a_{1_1}, \ldots, a_{1_n}>,...,<a_{m_1}, \ldots, a_{m_n}>>)$ (cond)
$= M(apndl:<<a_{1_1}, \ldots, a_{m_1}>,trans:<<a_{1_2}, \ldots, a_{1_n}>,...,<a_{m_2}, \ldots, a_{m_n}>>>)$
(constr,sel,tl,α)
$= M(apndl:<<a_{1_1}, \ldots, a_{m_1}>,<<a_{1_2}, \ldots, a_{m_2}>,...,<a_{1_n}, \ldots, a_{m_n}>>>)$ (trans)
$= <<a_{1_1}, \ldots, a_{m_1}>,...,<a_{1_n}, \ldots, a_{m_n}>>$ (apndl)

■

Theorem T12: $M(1^{st} \circ apndl) \sqsubseteq M(1^{st})$

Proof:

case 1: $x = \langle a,\phi \rangle$

$M(1^{st} \circ apndl{:}\langle a,\phi \rangle)$
$= \mathbf{M}(1^{st}){:}M(apndl{:}\langle a,\phi \rangle)$ (comp)
$= \mathbf{M}(1^{st}){:}\langle a \rangle$ (apndl)
$= \mathbf{a}$ (1^{st})
$M(1^{st}{:}\langle a,\phi \rangle = a$ (1^{st})

case 2: $x = \langle a,\langle b_1, \ldots, b_n \rangle \rangle$

$M(1^{st} \circ apndl{:}\langle a,\langle b_1, \ldots, b_n \rangle \rangle)$
$= \mathbf{M}(1^{st}){:}M(apndl{:}\langle a,\langle b_1, \ldots, b_n \rangle \rangle)$ (comp)
$= \mathbf{M}(1^{st}){:}\langle a,b_1, \ldots, b_n \rangle$ (apndl)
$= \mathbf{a}$ (1^{st})
$M(1^{st}{:}\langle a,\langle b_1, \ldots, b_n \rangle \rangle) = a$ (1^{st})

■

Theorem T13: $M(1^{st} \circ apndr) \sqsubseteq M(null \circ 1^{st} \rightarrow 2^{nd} ; 1^{st} \circ 1^{st})$

Proof:

case 1: $x = \langle \phi,b \rangle$

$M(1^{st} \circ apndr{:}\langle \phi,b \rangle)$
$= \mathbf{M}(1^{st}){:}M(apndr{:}\langle \phi,b \rangle)$ (comp)
$= \mathbf{M}(1^{st}){:}\langle b \rangle$ (apndr)
$= \mathbf{b}$ (1^{st})
$M((null \circ 1^{st} \rightarrow 2^{nd} ; 1^{st} \circ 1^{st}){:}\langle \phi,b \rangle)$
$= \mathbf{M}(2^{nd}{:}\langle \phi,b \rangle)$ (cond)
$= \mathbf{b}$ (2^{nd})

case 2: $x = \langle \langle a_1, \ldots, a_n \rangle,b \rangle$

$M(1^{st} \circ apndr{:}\langle \langle a_1, \ldots, a_n \rangle,b \rangle)$
$= \mathbf{M}(1^{st}){:}M(apndr{:}\langle \langle a_1, \ldots, a_n \rangle,b \rangle)$ (comp)
$= \mathbf{M}(1^{st}){:}\langle a_1, \ldots, a_n,b \rangle$ (apndr)
$= \mathbf{a_1}$ (1^{st})
$M((null \circ 1^{st} \rightarrow 2^{nd} ; 1^{st} \circ 1^{st}){:}\langle \langle a_1, \ldots, a_n \rangle,b \rangle)$
$= \mathbf{M}(1^{st} \circ 1^{st}{:}\langle \langle a_1, \ldots, a_n \rangle,b \rangle)$ (cond)
$= \mathbf{M}(1^{st}{:}\langle a_1, \ldots, a_n \rangle)$ (comp, 1^{st})
$= \mathbf{a1}$ (1^{st})

■

Theorem T14: $M(s^{th}) \sqsubseteq M((s-1)^{th} \circ tl)$

Proof:

$\mathbf{x} = \langle a_1, \ldots, a_n \rangle \quad n \geq s$

$\mathbf{M}(s^{th}{:}\langle a_1, \ldots, a_n \rangle) = a_s$ (s^{th})

$\mathbf{M}((s-1)^{th} \circ tl){:}\langle a_1, \ldots, a_n \rangle$

$= M((s-1)^{th}){:}M(tl{:}\langle a_1, \ldots, a_n \rangle)$ (comp)

$= M((s-1)^{th}){:}\langle a_1, \ldots, a_n \rangle$ (tl)

$= a_s$ (s^{th})

■

Theorem T15: $M(r1^{st} \circ apndl) \sqsubseteq M(null \circ 2^{nd} \rightarrow 1^{st} ; r1^{st} \circ 2^{nd})$

Proof:

case 1: $x = \langle a, \phi \rangle$

$\mathbf{M}(r1^{st} \circ apndl{:}\langle a, \phi \rangle)$

$= M(r1^{st}){:}M(apndl{:}\langle a, \phi \rangle)$ (comp)

$= M(r1^{st}){:}\langle a \rangle$ (apndl)

$= a$ ($r1^{st}$)

$\mathbf{M}((null \circ 2^{nd} \rightarrow 1^{st} ; r1st \circ 2^{nd}){:}\langle a, \phi \rangle)$

$= M(1^{st}{:}\langle a, \phi \rangle)$ (cond)

$= a$ (1^{st})

case 2: $x = \langle a, \langle b_1, \ldots, b_n \rangle \rangle$

$\mathbf{M}(r1^{st} \circ apndl{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$

$= M(r1^{st}){:}M(apndl{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (comp)

$= M(r1^{st}){:}\langle a, b_1, \ldots, b_n \rangle$ (apndl)

$= b_n$ ($r1^{st}$)

$\mathbf{M}((null \circ 2^{nd} \rightarrow 1^{st} ; r1^{st} \circ 2^{nd}){:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$

$= M(r1^{st} \circ .2nd{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (cond)

$= M(r1^{st}){:}M(2^{nd}{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (comp)

$= M(r1^{st}){:}\langle b_1, \ldots, b_n \rangle$ (2^{nd})

$= b_n$ ($r1^{st}$)

■

Theorem T16: $M(r1^{st} \circ apndr) \sqsubseteq M(2^{nd})$

Proof: similar to proof of T12

Theorem T17: $M(rs^{th}) \sqsubseteq M(r(s-1)^{th} \circ tlr)$

Proof: similar to proof of T14

Theorem T18: $M(/f) \sqsubseteq M(f \circ [1^{st}, /f \circ [2^{nd}, \ldots, n^{th}]])$ where n = length of x and n > 1

Proof:

$x = \langle a_1, \ldots, a_n \rangle$

$M(/f{:}\langle a_1, \ldots, a_n \rangle) = M(f){:}\langle a_1, M(/f){:}\langle a_2, \ldots, a_n \rangle \rangle$ (/)

$M(f \circ [1^{st}, /f \circ [2^{nd}, \ldots, n^{th}]]{:}\langle a_1, \ldots, a_n \rangle)$

$= M(f){:}M([1^{st}, /f \circ [2^{nd}, \ldots, n^{th}]{:}\langle a_1, \ldots, a_n \rangle$ (comp)

$= M(f){:}\langle a_1, M(f){:}\langle a_2, \ldots, a_n \rangle \rangle$ (constr,selectors,comp)

■

Theorem T19: $M(/f \circ apndl) \sqsubseteq M(null \circ 2^{nd} \rightarrow 1^{st} ; f \circ [1^{st}, /f \circ 2^{nd}])$

Proof:

case 1: $x = \langle a, \phi \rangle$

$M(/f \circ apndl{:}\langle a, \phi \rangle)$

$= M(/f){:}M(apndl{:}\langle a, \phi \rangle)$ (comp)

$= M(/f){:}\langle a \rangle$ (apndl)

$= a$ (/)

$M((null \circ 2^{nd} \rightarrow 1^{st} ; f \circ [1^{st}, /f \circ 2^{nd}]){:}\langle a, \phi \rangle)$

$= M(1^{st}{:}\langle a, \phi \rangle)$ (cond)

$= a$ (1^{st})

case 2: $x = \langle a, \phi \rangle$

$M(/f \circ apndl{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$

$= M(/f){:}M(apndl{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (comp)

$= M(/f){:}\langle a, b_1, \ldots, b_n \rangle$ (apndl)

$= M(f){:}\langle a, M(/f){:}\langle b_1, \ldots, b_n \rangle \rangle$ (/)

$M((null \circ 2^{nd} \rightarrow 1^{st} ; f \circ [1^{st}, /f \circ 2^{nd}]){:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle$

$= M(f \circ [1^{st}, /f \circ 2^{nd}]{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (cond)

$= M(f){:}M([1^{st}, /f \circ 2^{nd}]{:}\langle a, \langle b_1, \ldots, b_n \rangle \rangle)$ (comp)

$= M(f){:}\langle a, M(/f){:}\langle b_1, \ldots, b_n \rangle \rangle$ (constr,comp,selectors)

■

Theorem T20: $M(\alpha f \circ apndl) \sqsubseteq M(apndl \circ [f \circ 1^{st}, \alpha f \circ 2^{nd}])$

Proof:

case 1: $x = \langle a, \phi \rangle$

$M(\alpha f \circ apndl:\langle a,\phi\rangle)$
$= M(\alpha f):M(apndl:\langle a,\phi\rangle)$ (comp)
$= M(\alpha f):\langle a\rangle$ (apndl)
$= \langle M(f):a\rangle$ (α)
$M(apndl \circ [f \circ 1^{st}, \alpha f \circ 2^{nd}]:\langle a,\phi\rangle)$
$= M(apndl):M([f \circ 1^{st}, \alpha f \circ 2^{nd}]:\langle a,\phi\rangle)$ (comp)
$= M(apndl):\langle M(f):a, M(\alpha f):\phi\rangle$ (constr,selectors,comp)
$= M(apndl):\langle M(f):a,\phi\rangle$ (α)
$= \langle M(f):a\rangle$ (apndl)

case 2: $x = \langle a, \langle b_1, \ldots, b_n\rangle\rangle$

$M(\alpha f \circ apndl:\langle a, \langle b_1, \ldots, b_n\rangle\rangle)$
$= M(\alpha f):M(apndl:\langle a, \langle b_1, \ldots, b_n\rangle\rangle)$ (comp)
$= M(\alpha f):\langle a, b_1, \ldots, b_n\rangle$ (apndl)
$= \langle M(f):a, M(f):b_1, \ldots, M(f):b_n\rangle$ (α)
$M(apndl \circ [f \circ 1^{st}, \alpha f \circ 2^{nd}]:\langle a, \langle b_1, \ldots, b_n\rangle\rangle)$
$= M(apndl):M([f \circ 1^{st}, \alpha f \circ 2^{nd}]:\langle a, \langle b_1, \ldots, b_n\rangle\rangle)$ (comp)
$= M(apndl):\langle M(f):a, M(\alpha f):\langle b_1, \ldots, b_n\rangle\rangle$ (constr,selectors,comp)
$= M(apndl):\langle M(f):a, \langle M(f):b_1, \ldots, M(f):b_n\rangle\rangle$ (α)
$= \langle M(f):a, M(f):b_1, \ldots, M(f):b_n\rangle$ (apndl)

■

Theorem T21: $M(\alpha f \circ apndr) \sqsubseteq M(apndr \circ [\alpha f \circ 1^{st}, f \circ 2^{nd}])$

Proof: similar to proof of T19

Theorem T22: $M((bu\ f\ a) \circ g) \sqsubseteq M(f \circ [\bar{a}, g])$ where a is an object

Proof:

$M((bu\ f\ a) \circ g:x)$
$= M(bu\ f\ a):M(g:x)$ (comp)
$= M(f):\langle a, M(g:x)\rangle$ (bu)
$M(f \circ [\bar{a}, g]:x)$
$= M(f):M([\bar{a}, g]:x)$ (comp)
$= M(f):\langle a, M(g:x)\rangle$ (constr,constant)

■

Theorem T23: $M(/f) \sqsubseteq M(\text{null} \circ \text{tl} \rightarrow 1^{st} ; f \circ [1^{st}, / f \circ \text{tl}])$

Proof:

case 1: $x = \langle a \rangle$

$M(/f{:}\langle a \rangle) = a$ (/)

$M((\text{null} \circ \text{tl} \rightarrow 1^{st} ; f \circ [1^{st}, / f \circ \text{tl}]){:}\langle a \rangle)$

$= M(1^{st}{:}\langle a \rangle)$ (cond)

$= a$ (1^{st})

case 2: $x = \langle a_1, \ldots, a_n \rangle$

$M(/f{:}\langle a_1, \ldots, a_n \rangle) = M(f){:}\langle a, M(/f){:}\langle a_2, \ldots, a_n \rangle \rangle$ (/)

$M((\text{null} \circ \text{tl} \rightarrow 1^{st} ; f \circ [1^{st}, / f \circ \text{tl}]){:}\langle a_1, \ldots, a_n \rangle)$

$= M(f \circ [1^{st}, / f \circ \text{tl}]{:}\langle a_1, \ldots, a_n \rangle)$ (cond)

$= M(f){:}M([1^{st}, / f \circ \text{tl}]{:}\langle a_1, \ldots, a_n \rangle)$ (comp)

$= M(f){:}\langle a_1, M(/f){:}\langle a_2, \ldots, a_n \rangle \rangle$ (constr, 1^{st}, tl, comp)

■

APPENDIX D

Proofs that the important laws of the Algebra of FP Programs still hold when *apply* is added to FP:

$$h \circ (p \rightarrow f ; g) \stackrel{?}{=} p \rightarrow h \circ f ; h \circ g$$

apply ∘ (p → f ; g):x
⇒ apply:((p → f ; g):x) (comp)
⇒ apply:(p:x) → (f:x) ; (g:x) (cond)
case (p:x) = T
⇒ apply:(f:x) (cond)
case (p:x) = F
⇒ apply:(g:x) (cond)
otherwise
⇒ ⊥ (cond)

(p → apply ∘ f ; apply ∘ g):x
⇒ (p:x) → (apply ∘ f:x) ; (apply ∘ g:x) (cond)
⇒ (p:x) → (apply:(f:x)) ; (apply:(g:x)) (comp)
case (p:x) = T
⇒ apply:(f:x) (cond)
case (p:x) = F
⇒ apply:(g:x) (cond)
otherwise
⇒ ⊥ (cond)

$$(p \rightarrow f ; g) \circ h \stackrel{?}{=} p \circ h \rightarrow f \circ h ; g \circ h$$

(p → f ; g) ∘ apply:<h,y>
⇒ (p → f ; g):(apply:<h,y>) (comp)
⇒ (p → f ; g):(h:y) (apply)
⇒ (p:(h:y)) → (f:(h:y)) ; (g:(h:y)) (cond)
case p:(h:y) = T
⇒ f:(h:y) (cond)
case p:(h:y) = F

```
          ⇒ g:(h:y)                                          (cond)
      otherwise
          ⇒ ⊥                                                (cond)

(p ∘ apply → f ∘ apply ; g ∘ apply):<h,y>
  case p ∘ apply:<h,y> = p:(h:y) = T
      ⇒ f ∘ apply:<h,y>                                      (cond)
      ⇒ f:(apply:<h,y>)                                      (comp)
      ⇒ f:(h:y)                                              (apply)
  case p ∘ apply:<h,y> = p:(h:y) = F
      ⇒ g ∘ apply:<h,y>                                      (cond)
      ⇒ g:(apply:<h,y>)                                      (comp)
      ⇒ g:(h:y)                                              (apply)
  otherwise
      ⇒ ⊥                                                    (cond)
```

Proofs that the important laws of the Algebra of FP Programs hold when Y is added to FP:

$$[f,g] \circ h \stackrel{?}{=} [f \circ h, g \circ h]$$

$[f,g] \circ Ye:x$

$\Rightarrow [f,g]:(Ye:x)$ (comp)

$\Rightarrow \langle f:(Ye:x), g:(Ye:x)\rangle$ (constr)

$[f \circ Ye, g \circ Ye]:x$

$\Rightarrow \langle f \circ Ye:x, g \circ Ye:x\rangle$ (constr)

$\Rightarrow \langle f:(Ye:x), g:(Ye:x)\rangle$ (comp)

$$(p \rightarrow f ; g) \circ h \stackrel{?}{=} (p \circ h) \rightarrow (f \circ h) ; (g \circ h)$$

$(p \rightarrow f ; g) \circ Ye:x$

$\Rightarrow (p \rightarrow f ; g):(Ye:x)$ (comp)

$\Rightarrow (p:(Ye:x)) \rightarrow (f:(Ye:x)) ; (g:(Ye:x))$ (cond)

casep:$(Ye:x)$ = T

$\Rightarrow f:(Ye:x)$ (cond)

case p:$(Ye:x)$ = F

$\Rightarrow g:(Ye:x)$ (cond)

otherwise

$\Rightarrow \bot$ (cond)

$(p \circ Ye \rightarrow f \circ Ye ; g \circ Ye):x$

$\Rightarrow (p \circ Ye:x) \rightarrow (f \circ Ye:x) ; (g \circ Ye:x)$ (cond)

case $(p \circ Ye:x) = (p:(Ye:x))$ = T

$\Rightarrow f \circ Ye:x$ (cond)

$\Rightarrow f:(Ye:x)$ (comp)

case $(p \circ Ye:x) = (p:(Ye:x))$ = F

$\Rightarrow g \circ Ye:x$ (cond)

$\Rightarrow g:(Ye:x)$ (comp)

otherwise

$\Rightarrow \bot$ (cond)

$$h \circ (p \rightarrow f ; g) \stackrel{?}{=} p \rightarrow (h \circ f) ; (h \circ g)$$

$Ye \circ (p \rightarrow f ; g):x$

$\Rightarrow$ $\curlyvee$e:((p $\rightarrow$ f ; g):x) (comp)
$\Rightarrow$ $\curlyvee$e:((p:x) $\rightarrow$ (f:x) ; (g:x)) (cond)
casep:x = T
$\Rightarrow$ $\curlyvee$e:(f:x) (cond)
casep:x = F
$\Rightarrow$ $\curlyvee$e:(g:x) (cond)
otherwise
$\Rightarrow$ $\curlyvee$e:$\perp$ (cond)
$\Rightarrow$ $\perp$

p $\rightarrow$ $\curlyvee$e $\circ$ f ; $\curlyvee$e $\circ$ g:x $\Rightarrow$ (p:x) $\rightarrow$ ($\curlyvee$e $\circ$ f:x) ; ($\curlyvee$e $\circ$ g:x) (cond)
case (p:x) = T
$\Rightarrow$ $\curlyvee$e $\circ$ f:x (cond)
$\Rightarrow$ $\curlyvee$e:(f:x) (comp)
case (p:x) = F
$\Rightarrow$ $\curlyvee$e $\circ$ g:x (cond)
$\Rightarrow$ $\curlyvee$e:(g:x) (comp)
otherwise
$\Rightarrow$ $\perp$ (cond)

APPENDIX E

Proof that τ, a function that maps L_{APP}, a lanaguage of applications, into L_{AFP}, an AFP language, preserves the meaning of its domain.

Definition: L_{APP} is the language of expressions *e* such that

1) $e \in L_{AFP}$
2) $e = (c:d)$ and $c \in L_{APP}$ and $d \in L_{APP}$
3) Nothing else is in L_{APP}

Definition: $\mu : (L_{AFP} + L_{APP}) \rightarrow \Lambda$

(m1) $\mu(0) = 0_\sigma$ (0_σ is the object denoted by 0)
(m2) $\mu(a:b) = \mu(a):\mu(b)$
(m3) $\mu(f \circ g) = \lambda 0 . \mu(f):(\mu(g):0)$
(m4) $\mu([f_1, \ldots, f_2]) = \lambda 0 . <\mu(f_1):0, \ldots, \mu(f_n):0>$
(m5) $\mu(\text{apply}) = \lambda<f,x> . f:x$
(m6) $\mu(f) = F(f)\phi$

f is a functional expression (i.e. an AFP expression that denotes a function). *F* is a semantic function that maps functional expressions into lambda expressions (Appendix B).

Definition: $\tau : L_{APP} \rightarrow L_{AFP}$

(b1) $\tau(a) = a$ ($a \in O$)
(b2) $\tau(a:b) = a:b$ ($a,b \in O$)
(b3) $\tau(a:(b:c)) = \tau(a \circ b:c)$ ($a,b \in O$)
(b4) $\tau(a:(b:c)) = \tau(a:\tau(b:c))$ ($a \in O$, b not $\in O$)
(b5) $\tau((a:b):c) = \tau(\text{apply} \circ [a,\text{Const}(\tau(c))]:b)$ ($a \in O$)

(b6) $\tau((a:b):c) = \tau(\tau(a:b):c)$ $\quad (a \notin O)$

(c1) $Const(a) = \bar{a}$ $\quad (a \in O)$
(c2) $Const(a:b) = a \circ Const(b)$ $\quad (a \in O)$

Definition: $N : L_{AFP} + L_{APP} \rightarrow$ integer

(n1) $N(a) = 0$ $\quad (a \in$ (objects + functional expressions))
(n2) $N(a:b) = N(a) + N(b) + 1$

Theorem $\forall\ x \in L_{APP}, \mu(x) = \mu(\tau(x))$

Proof: by Induction on N(x)

Bases: I) $N(x) = 0$

1) $\tau(x) = x$	(b1)
2) $\mu(x) = \mu(\tau(x))$	(1)

II) $N(x) = 1$

1) $\tau(x) = \tau(a:b) = a:b$	(b2)
2) $\mu(x) = \mu(a:b) = \mu(\tau(x))$	(1)

Hypothesis: $\forall$ x in L_{APP} s.t. $N(x) < n$, $\mu(x) = \mu(\tau(x))$

Induction: $\forall$ x in L_{APP} s.t. $N(x) = n$, $\mu(x) = \mu(\tau(x))$

Case 1: $x = a:(b:c)$ $\quad a,b \in$ objects + functional expressions

1) $\mu(x) = \mu(a:(b:c)) = \mu(a):\mu(b:c)$	(m2)
2) $= \mu(a):(\mu(b):\mu(c))$	(m2)
3) $\mu(\tau(x)) = \mu(\tau(a:(b:c))) = \mu(a \circ b:c)$	(b3)
4) $= \mu(a \circ b):\mu(c)$	(m2)
5) $= (\lambda O\,.\,\mu(a):(\mu(b):O)):\mu(c)$	(m3)
6) $= \mu(a):(\mu(b):\mu(c))$	(beta red)
7) $= \mu(x)$	(2)

Case 2: x = a:(b:c) a ∈ (objects + functional expressions), b ∉ (objects + functional expressions)

1) $\mu(\tau(x)) = \mu(\tau(a:(b:c))) = \mu(\tau(a:\tau(b:c)))$ (b4)
2) $N(b:c) = N(b) + N(c) + 1$ (N)
3) $N(\tau(b:c)) \leq 1$ (Lemma 1)
4) $N(\tau(b:c)) < N(b:c)$ (2 & 3)
5) $N(a:\tau(b:c)) < N(a:(b:c)) = n$ (4)
6) $\mu(a:\tau(b:c)) = \mu(\tau(a:\tau(b:c)))$ (hyp)
7) $\mu(b:c) = \mu(\tau(b:c))$ (hyp)
8) $\mu(a:\tau(b:c)) = \mu(a):\mu(\tau(b:c))$ (m2)
9) $= \mu(a):\mu(b:c)$ (7)
10) $\mu(a):\mu(b:c) = \mu(\tau(a:\tau(b:c)))$ (6,8,9)
11) $\mu(x) = \mu(a:(b:c)) = \mu(a):\mu(b:c)$ (m2)
12) $\mu(x) = \mu(\tau(x))$ (11,10)

Case 3: x = (a:b):c a ∈ (objects + functional expressions)

1) $\mu(x) = \mu((a:b):c) = \mu(a:b):\mu(c)$ (m2)
2) $= (\mu(a):\mu(b)):\mu(c)$ (m2)
3) $\tau(x) = \tau((a:b):c) = \tau(apply \circ [a,Const(\tau(c))]:b)$ (b5)
4) $N(\tau(c)) \leq 1$ (Lemma 1)
5) $Const(\tau(c))$ is an FP function (Const)
6) $apply \circ [a,Const(\tau(c))] \in$ (objects+f–exprs) (5 & def of FP function)
7) $n = N(x) = N(a) + N(b) + N(c) + 2 = N(b) + N(c) + 2$ (N)
8) $N(apply \circ [a,Const(\tau(c))]:b) = N(b) + 1 < n$ (N)
9) $\mu(apply \circ [a,Const(\tau(c))]:b) = mu(\tau(apply \circ [a,Const(\tau(c))]:b))$ (hyp)
10) $\mu(apply \circ [a,Const(\tau(c))]:b = \mu(apply \circ [a,Const(\tau(c))]):\mu(b)$ (m2)
11) $= (\lambda 0.\mu(apply):(\mu([a,Const(\tau(c))]):0)):\mu(b)$ (m2)
12) $= \mu(apply):(\mu([a,Const(\tau(c))]):u(b))$ (beta red)
13) $= \mu(apply):((\lambda 0.<\mu(a):0,\mu(Const(\tau(c))):0>):\mu(b))$ (m4)
14) $= \mu(apply):((\lambda 0.<\mu(a):0,(\lambda 0.\mu(\tau(c)):0>):\mu(b))$ (Lemma 2)
15) $= \mu(apply):(<\mu(a):\mu(b),(\lambda 0.\mu(\tau(c)):\mu(b)>)$ (beta red)
16) $= \mu(apply):(<\mu(a):\mu(b),\mu(\tau(c))>)$ (beta red)
17) $= \mu(apply):(<\mu(a):\mu(b),\mu(c)>)$ (hyp)
18) $= (\lambda<f,x> \circ f:x):(<\mu(a):\mu(b),\mu(c)>)$ (m5)
19) $= (\mu(a):\mu(b)):\mu(c)$ (beta red)
20) $\mu(x) = \mu(\tau(x))$ (2,3,9,19)

Case 4: x = (a:b):c a ∉ (objects + functional expressions)

1) $\mu(a:b) = \mu(\tau(a:b))$ (hyp)
2) $N(a:b) = N(a) + N(b) + 1$ (N)
3) $N(\tau(a:b)) \leq 1$ (Lemma 1)
4) $N(a) \geq 1$ (a not in (objects + f-exprs)
5) $N(a:b) \geq N(b) + 2$ (N,4)
6) $N(\tau(a:b)) < N(a:b)$ (3,5)
7) $N(\tau(a:b):c) < N((a:b):c) = n$ (6)
8) $\mu(\tau(a:b):c) = \mu(\tau(\tau(a:b):c))$ (hyp)
9) $\mu(\tau(x)) = \mu(\tau(\tau(a:b):c)) = \mu(\tau(a:b):c)$ (b6)
10) $= \mu(\tau(a:b)):\mu(c)$ (m2)
11) $= \mu(a:b):\mu(c)$ (1)
12) $\mu(x) = \mu((a:b):c) = \mu(a:b):\mu(c)$ (m2)
13) $= \mu(\tau(x))$ (10)

Lemma 1: $\forall$ a ∈ L_{APP}, $N(\tau(a)) \leq 1$

Proof: by Induction on N(a)

Bases: I) a ∈ (objects + functional expressions)
$N(\tau(a)) = N(a)$ (b1)
$N(a) = 0$ (N)

II) a = p:q p,q ∈ (objects + functional expressions)
$N(\tau(a)) = N(p:q) = N(p) + N(q) + 1$ (N)
$= 0 + 0 + 1$ (N)
$= 1$

Hypothesis: $\forall$ a ∈ L_{APP} s.t. $N(a) < n$, $N(\tau(a)) \leq 1$

Induction: $\forall$ a ∈ L_{APP} s.t. $N(a) = n$, $N(\tau(a)) \leq 1$

Case 1: a = p:(q:r) p,q ∈ (objects + functional expressions)

1) $N(\tau(a)) = N(\tau(p:(q:r))) = N(\tau(p \circ q:r))$ (b3)
2) $n = N(p:(q:r)) = N(r) + 2$ (N)
3) $N(p \circ q:r) = N(r) + 1 < n$ (N)
4) $N(\tau(p \circ q:r)) \leq 1$ (hyp)
5) $N(\tau(a)) \leq 1$

Case 2: a = p:(q:r) p ∈ (objects + functional expressions), q ∉ (objects + functional expressions)

1) $N(\tau(a)) = N(\tau(p:(q:r)) = N(\tau(p:(\tau(q:r)))$ (b4)
2) $n = N(p:(q:r)) = N(q) + N(r) + 2$ (N)
3) $N(q:r) = N(q) + N(r) + 1 < n$ (N)
4) $N(\tau(q:r)) \leq 1$ (hyp)
5) $N(p:\tau(q:r)) \leq 2 < n$ (N)
6) $N(\tau(p:\tau(q:r))) \leq 1$ (hyp)
7) $N(\tau(a)) \leq 1$ (1,6)

Case 3: a = ((p:q):r p ∈ (objects + functional expressions)

1) $N(\tau(a)) = N(\tau((p:q):r)) = N(\tau(\text{apply} \circ [\mathbf{p},\text{Const}(\tau(r))]:q))$ (b5)
2) $N(\tau(r)) \leq 1$ (hyp)
3) $\text{Const}(\tau(r))$ is an FP function (Const)
4) $\text{apply} \circ [p,\text{Const}(\tau(r))] \in j$ (def of FP function)
5) $n = N((p:q):r) = N(q) + N(r) + 2$ (N)
6) $N(\text{apply} \circ [p,\text{Const}(\tau(r))]:q = N(q) + 1 < n$ (N)
7) $N(\tau(\text{apply} \circ [p,\text{Const}(\tau(r))]:q)) \leq 1$ (hyp)
8) $N(\tau(a)) \leq 1$ (1,7)

Case 4: a = (p:q):r p ∉ (objects + functional expressions)

1) $N(\tau(a)) = N(\tau(\tau(p:q):r))$ (b6)
2) $n = N((p:q):r) = N(p) + N(q) + N(r) + 2$ (N)
3) $N(p:q) = N(p) + N(q) + 1 < n$ (N)
4) $N(\tau(p:q)) \leq 1$ (hyp)
5) $N(\tau(p:q):r) \leq 1 + N(r) + 1 < n$ (4,N)
6) $N(\tau(\tau(p:q):r) \leq 1$ (hyp)
7) $N(\tau(a)) \leq 1$

Lemma 2: $\forall$ $a \in L_{AFP}, \mu(\text{Const}(a)) = \lambda 0.\mu(a)$

Case 1: a ∈ (objects + functional expressions)

$\mu(\text{Const}(a)) = u(\bar{a}) = \lambda 0.\mu(a)$ (μ)

Case 2: a = p:q p,q ∈ (objects + functional expressions)

$\mu(\text{Const}(a)) = \mu(\text{Const}(p:q)) = u(p \circ \bar{q})$ (Const)
$= \lambda 0.\mu(p):(\mu(\bar{q}):0)$ (m3)

$= \lambda 0.\mu(p):((\lambda 0.\mu(q)):0)$ (m6,F)
$= \lambda 0.\mu(p):\mu(q)$ (beta red)
$= \lambda 0.\mu(p:q)$ (m2)
$= \lambda 0.\mu(a)$

APPENDIX F

THE SYNTAX OF FP

The syntax of the user-level source language is defined by a context-free grammar. Annotations have been added to this grammar to describe the translation of the source language to AST's. The Syntax Analyzer, which performs the translation, was automatically produced by *sac*, a syntax analyzer constructer. [1]

Concrete Grammar

```
program ::= statement-list                                  $Opr program
statement-list ::= statement statement-list
                 | ε
statement ::= definition                                    $Opr def
            | application
definition ::= def Identifier ≡ functional-form
functional-form ::= term ff-tail
ff-tail ::= → functional-form ; functional-form             $Opr condition (1)
          | ε
term ::= factor term-tail
term-tail ::= ∘ term                                        $Opr compose (1)
            | ε
factor ::= insert-L
         | apply-to-all
         | ff
ff ::= construction
     | constant
     | binary-to-unary
     | while
     | (functional-form)
```

```
        | function-name
function-name ::= selector                                  $Opr select
                | tl
                | id
                | atom
                | eq
                | null
                | reverse
                | distl
                | distr
                | length
                | add
                | sub
                | mul
                | div
                | trans
                | and
                | or
                | not
                | apndl
                | apndr
                | right-selector                            $Opr Rselect
                | tlr
                | rotl
                | rotr
                | Identifier
                | gt
                | ge
                | lt
                | le
                | ne
selector ::= 1st | 2nd | 3rd | ...
right-selector ::= r 1st | 2nd | 3rd | ...
construction ::= [functional-list]                          $Opr construct
functional-list ::= functional-form functional-list-tail
                  | ε
functional-list-tail ::= ,functional-form functional-list-tail
                       | ε
constant ::= object                                         $Opr constant
insertL ::= /factor                                         $Opr insertL
apply-to-all ::= αfactor                                    $Opr applyAll
```

```
binary-to-unary ::= bu functional-form object                    $Opr bu
while ::= while functional-form functional-form                  $Opr while
application ::= functional-form : fp-object                      $Opr apply
fp-object ::= object
            | application
object ::= <sequence>                                           $Opr sequence
         | atom
         | bottom
sequence ::= object s-tail
           | ε
s-tail ::= ,object s-tail
         | ε
atom ::= literal
       | number
       | boolean
bottom ::= ⊥
literal ::= Literal
number ::= Numeric
boolean ::= T
          | F
```

Concessions to the ASCII Character Set

In order to make it possible to input FP expressions using the ASCII character set, the following substitutions are made:

FP symbol	ASCII equivalent
∘	.
1^{st},2^{nd},...	$1,$2,...
r1^{st},r2^{nd},...	$r1,$r2,...
α	@
→	->
ā	"a"

APPENDIX G

ABSTRACT SYNTAX GRAMMAR FOR FP

Syntactic Domains:

atom ={Numeral,Literal}

definition ={def}

application ={apply}

object ={atom,sequence,⊥}

f-expr = {condition, compose, select, Rselect, construct, constant, insertR, insertL, applyAll, bu, while, tl, id, atom, eq, null, reverse, distl, distr, length, add, sub, mul, div, trans, and, or, not, apndl, apndr, Identifier}

Productions:

program → (definition + application)*

def → Identifier f-expr

apply → f-expr (object + suspension + apply)

condition → f-expr f-expr f-expr

compose → f-expr f-expr

select → Numeral

Rselect → Numeral

construct → f-expr *

constant → object

insertL → f-expr

applyAll → f-expr

bu → f-expr object

while → f-expr f-expr

sequence → object *

BIOGRAPHICAL NOTE

The author was born 19 August 1952, in Bayport, New York. She attended the Bayport public schools and graduated from James Wilson Young High School in 1969.

The author attended Indiana University School of Music, receiving a Bachelor of Music in 1973 and Master of Music in 1976. During that time she taught piano and was an associate instructor in Music Theory. In 1978-1979 the author was a visiting lecturer at Indiana University Southeast.

In 1980 the author began taking courses in Computer Science at S.U.N.Y. at Stony Brook, where she married Eugene Rollins. In 1981 she began her studies at the Oregon Graduate Center where she completed the requirements for the degree Master of Science in Computer Science, in August 1983.