FORMAL REPRESENTATIONS FOR RECURSIVELY DEFINED FUNCTIONAL PROGRAMS

John H. Williams

IBM Research Laboratory
5600 Cottle Road
San Jose, California 95193

Introduction

In [Ba 78] Backus shows how a formal system for functional programming (FFP) can be used to represent all of the objects, primitive functions, and combining forms contained in his informal system of functional programming (FP). In particular, using the rule of *metacomposition* from his definition of the meaning function μ of the FFP system, he shows how each of the combining forms of FP can be realized as sequences in FFP. He also states, "in addition to its use in defining functional forms, metacomposition can be used to create recursive functions directly without the use of recursive definitions of the form"

Def $f = Ef$,

and he shows by way of example that the object $\langle LAST \rangle$ will correctly represent the FP function defined by:

Def last = null∘tl → 1; last∘tl

if $LAST$ is taken to be the FFP object that represents the (nonrecursive) function

null∘tl∘2 → 1∘2; APPLY∘[1, tl∘2] .

One can verify that the meaning of $\langle LAST \rangle$ applied to the sequence $\langle A, B \rangle$ is indeed B, *i.e.* that

$$\mu(\langle LAST \rangle : \langle A, B \rangle) = B$$

and in general that

$$\mu(\langle LAST \rangle : \langle x_1, \ldots, x_n \rangle) = x_n$$

when $\langle x_1, \ldots, x_n \rangle$ is a constant expression of FFP.

It is remarkable that by using such a simple device as metacomposition Backus was able to represent such recursively defined functions without relying on the existence of the (rather complicated) Y combinator of [Ch 41] and [Cu 58] or introducing into his system an additional mechanism to maintain an environment of name/value bindings and a special "labeling" form as was done in Lisp [Mc 60]. In fact, so powerful is the added environment of Lisp that the introduction of the special form LABEL is actually unnecessary. To represent the function defined by

DEFINE (LAST (λ (X) E_{LAST}))

we can use the pure Lisp expression

(λ (X) ((λ (LAST) (LAST X)) (QUOTE (λ (X) E_{LAST}))))

instead of the special form

(LABEL LAST (λ (X) E_{LAST})) .

So it is noteworthy that metacomposition enables us to represent the function "last" without the use of any added environment-maintaining mechanism.

However, the defining equation for "last" is an example of a particularly simple class of recursive forms -- the so called "tail recursions", and it remains to be shown that the existence of the FFP object <*LAST*> is not just due to some special properties of tail-recursive equations, but rather that all FP functions can be so represented in FFP.

In this note we give a general algorithm for producing FFP representatives for arbitrary, recursively defined FP functions. In doing so, we will have demonstrated the universality of FFP, since all partial recursive functions can be defined in FP. (That all primitive recursive functions can be defined is clear. The *min* operator μx:R(x) of [KL 52] can be realized by the defined function

Def least = R → id; least ∘ + ∘ [id, $\overline{1}$]

since then μx:R(x) = least ∘ $\overline{0}$.) We will first give the algorithm for functions defined by a single equation and then generalize it to include functions defined by sets of mutually recursive equations.

Functions defined by a single equation

In order to show that every function definable in the informal FP system can be represented in FFP, we must give some representation-producing mechanism π that maps the domain **F** of expressions denoting functions in FP into the domain of FFP expressions such that

$\forall f$ in **F**, $\rho(\pi(f)) = f$.

Note that π will be a purely syntactic mapping from expressions in FP to expressions in FFP.

Now there are three types of functions in the FP system:

1) primitives **P**,
2) the closure of **P** under the combining forms, and
3) functions defined by (recursive) functional equations.

For functions of type 1) and 2) we will define π to be consistent with the representation Backus chose in [Ba 78]. For example,

π(tl) = TAIL
π(1) = 1
π(null ∘ tl) = <COMP, NULL, TAIL>
π([id, αtl]) = <CONS, ID, <ALPHA, TAIL>>
$\pi(\overline{3})$ = <CONST, 3>
etc.

We will make one modification to the representation of [Ba 78]. Since the sequence constructor of FFP is $\perp$-preserving, we cannot use <CONST, x> to represent $\overline{x}$ for all objects x. Notice that while the object <CONST, $\perp$> *will* correctly represent the function $\overline{\perp}$, a function such as

zerofilter = eq0 → id; $\overline{\perp}$

is not correctly represented by the FFP object

<COND, EQ0, ID, <CONST, $\perp$>>

since the presence of $\perp$ reduces that object to $\perp$. In this paper we will use the object UNDEF to represent the FP function $\overline{\perp}$.

For functions defined by a recursive equation we define π as follows:

Definition If $f = Ef$ is a recursively defined FP function, where E is built up by applying combining forms to primitive functions and the function letter f, then $\pi(f)$ is defined to be the FFP object $<\pi_f(Ef)>$ where $\pi_f(Ef)$ is the FFP object that results from applying the following representation-producing algorithm π_f to the form Ef.

In describing this algorithm we will use a shorthand notation to clarify the exposition. Since we know that all primitives and their closures under the combining forms have representing expressions in FFP, we will frequently use a more readable FP-like notation to denote those FFP representations. For example, instead of <COMP, TAIL, 2> we will write tl∘2, and instead of <COMP, APPLY, <CONS, 1, 2>> we will write APPLY∘[1, 2].

Algorithm (Representation-producing) The FFP representation of a form Ef with respect to the function letter f, $\pi_f(Ef)$, is constructed recursively by cases on the structure of Ef:

Form of Ef	FFP representation of Ef
p (p in **P**)	$p \circ 2$
f	APPLY
$\overline{x}$	$\overline{x}$ if $x \neq \perp$, else UNDEF
$Pf \rightarrow Qf; Rf$	$\pi_f(Pf) \rightarrow \pi_f(Qf); \pi_f(Rf)$
$[Pf, Qf]$	$[\pi_f(Pf), \pi_f(Qf)]$
$Pf \circ Qf$	$\pi_f(Pf) \circ [1, \pi_f(Qf)]$
$\alpha(Pf)$	$\alpha(\pi_f(Pf)) \circ$ distl

To illustrate the use of this algorithm, recall the definition

Def last = null∘tl → 1; last∘tl .

Then the FFP object representing last is <*LAST*> where

LAST

$= \pi_{last}(E(last))$

$= \pi_{last}(null \circ tl \rightarrow 1;\ last \circ tl)$

$= \pi_{last}(null \circ tl) \rightarrow \pi_{last}(1);\ \pi_{last}(last \circ tl)$

$= \pi_{last}(null) \circ [1, \pi_{last}(tl)] \rightarrow \pi_{last}(1);\ \pi_{last}(last \circ tl)$

$= (null \circ 2) \circ [1, tl \circ 2] \rightarrow 1 \circ 2;\ \pi_{last}(last) \circ [1, \pi_{last}(tl)]$

$= null \circ 2 \circ [1, tl \circ 2] \rightarrow 1 \circ 2;\ APPLY \circ [1, tl \circ 2]$

$= null \circ tl \circ 2 \rightarrow 1 \circ 2;\ APPLY \circ [1, tl \circ 2]$, since $defined \circ 2 \Rightarrow defined \circ 1$,

which is exactly the expression chosen by Backus in [Ba 78].

To prove that $<\pi_f(Ef)>$ always correctly represents the function f defined by

Def $f \equiv Ef$

we must show that

$$\rho(<\pi_f(Ef)>) = f ,$$

i.e. that for all objects x

$$\mu(<\pi_f(Ef)>:x) = f:x$$

or equivalently (according to the definition of metacomposition) that

$$\mu(\pi_f(Ef):<<\pi_f(Ef)>, x>) = f:x .$$

Theorem. If f is the FP function defined by **Def** f = Ef, then $\rho(<\pi_f(Ef)>) = f$.

Proof: We will show for all objects x and FP expressions Hf (*i.e.* functions built up from the primitives and the function f by the application of combining forms) that

(1) $\mu(\pi_f(Hf):<<\pi_f(Ef)>, x>) = Hf:x$

and therefore, *a fortiori*, that

$$\mu(\pi_f(Ef):<<\pi_f(Ef)>, x>) = Ef:x = f:x .$$

Note that when $x = \perp$, (1) holds trivially, since all functions are strict and the sequence constructor is $\perp$-preserving. For $x \neq \perp$, we will prove (1) by computational induction on the length of the sequence of reductions $(Hf:x \rightarrow \ldots \rightarrow y)$ that transforms the FP application $Hf:x$ into its meaning y. We will use the inductive hypothesis:

(2) If $Hf:x \rightarrow y$ in less than or equal to i steps, then $\mu(\pi_f(Hf):<<\pi_f(Ef)>, x>) = y$.

By a step we mean one of the simplifications given by Backus in his informal description of the semantics of FP systems [Ba78]. For example, the application of a primitive to an

object ($p{:}x \to y$) and the substitution of a definition for a defined function name ($\mathrm{f}{:}x \to \mathrm{Ef}{:}x$) are each considered to be one step in a reduction sequence. The number of steps in the sequence ($(\mathrm{Pf} \to \mathrm{Qf};\ \mathrm{Rf}){:}x \to \ldots \to y$) will be the number of steps to evaluate $\mathrm{Pf}{:}x$ plus the number to evaluate whichever branch of the conditional is subsequently taken, $\mathrm{Qf}{:}x$ or $\mathrm{Rf}{:}x$.

Basis, $i = 1$: If $\mathrm{Hf}{:}x \to y$ in one step, then there are two possible cases to consider:

(Note: In treating these cases it will be important to distinguish between FP and FFP expressions. At the same time, for ease of reading we will continue to use the FP-like abbreviations for FFP expressions. No confusion will arise if we remember that π_f maps FP expressions to FFP expressions and that the domain of μ is FFP expressions. To help clarify these notational conventions, in the proof of the first case we will show parenthetically the equivalent, unabbreviated forms of the FFP expressions being used.)

Case B1: $\mathrm{Hf} = p$ (where p is some primitive), and $p{:}x \to y$.

$\mu(\pi_f(\mathrm{Hf}){:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \mu(\pi_f(p){:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \mu(p \circ 2{:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$, (*i.e.* $\mu({<}\mathrm{COMP},P,2{>}{:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$, where $\rho(P) = p$)

$= \mu(p{:}x)$, (*i.e.* $\mu(P{:}x)$), since ${<}\pi_f(\mathrm{Ef}){>} \neq \perp$

$= p{:}x$, since $\rho(P) = p$

$= y$.

Case B2: $\mathrm{Hf} = \bar{y}$, and $\bar{y}{:}x \to y$.

$\mu(\pi_f(\mathrm{Hf}){:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \mu(\pi_f(\bar{y}){:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \mu(\bar{y}{:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \bar{y}{:}x$, since ${<}\pi_f(\mathrm{Ef}){>} \neq \perp$

$= y$.

Induction step: Suppose that $\mathrm{Hf}{:}x \to y$ in $\leq i{+}1$ steps.

Case I1. $\mathrm{Hf} = \mathrm{f}$, and $\mathrm{f}{:}x \to y$ in $\leq i{+}1$ steps. Then $\mathrm{Ef}{:}x \to y$ in $\leq i$ steps, so by (2)

$\mu(\pi_f(\mathrm{Ef}){:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>}) = y$.

Therefore, $\mu(\pi_f(\mathrm{Hf}){:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \mu(\pi_f(\mathrm{f}){:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \mu(\mathrm{APPLY}{:}{<}{<}\pi_f(\mathrm{Ef}){>}, x{>})$

$= \mu({<}\pi_f(\mathrm{Ef}){>}{:}x)$

$= \mu(\pi_f(\mathrm{Ef}):<<\pi_f(\mathrm{Ef})>, x>)$

$= y$.

Case I2. $\mathrm{Hf} = [\mathrm{If}, \mathrm{Jf}]$, and $[\mathrm{If}, \mathrm{Jf}]:x \to y$ in $\leq i+1$ steps. Then

$\mathrm{If}:x \to y_1$ in $\leq i$ steps
and $\mathrm{Jf}:x \to y_2$ in $\leq i$ steps,

where $y = <y_1, y_2>$, so that

$\mu(\pi_f(\mathrm{If}):<<\pi_f(\mathrm{Ef})>, x>) = y_1$
and $\mu(\pi_f(\mathrm{Jf}):<<\pi_f(\mathrm{Ef})>, x>) = y_2$.

Therefore, $\mu(\pi_f(\mathrm{Hf}):<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu(\pi_f([\mathrm{If}, \mathrm{Jf}]):<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu([\pi_f(\mathrm{If}), \pi_f(\mathrm{Jf})]:<<\pi_f(\mathrm{Ef})>, x>)$

$= <\mu(\pi_f(\mathrm{If}):<<\pi_f(\mathrm{Ef})>, x>) , \mu(\pi_f(\mathrm{Jf}):<<\pi_f(\mathrm{Ef})>, x>)>$

$= <y_1 , y_2>$

$= y$.

Case I3. $\mathrm{Hf} = \mathrm{Pf} \to \mathrm{Qf}; \mathrm{Rf}$, and $(\mathrm{Pf} \to \mathrm{Qf}; \mathrm{Rf}):x \to y$ in $\leq i+1$ steps.

similar to Case I2.

Case I4. $\mathrm{Hf} = \mathrm{If} \circ \mathrm{Jf}$, and $\mathrm{If} \circ \mathrm{Jf}:x \to y$ in $\leq i+1$ steps. Then $\exists z$ such that

$\mathrm{Jf}:x \to z$ in $\leq i$ steps
and $\mathrm{If}:z \to y$ in $\leq i$ steps,

so that

$\mu(\pi_f(\mathrm{Jf}):<<\pi_f(\mathrm{Ef})>, x>) = z$
and $\mu(\pi_f(\mathrm{If}):<<\pi_f(\mathrm{Ef})>, z>) = y$.

Therefore, $\mu(\pi_f(\mathrm{Hf}):<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu(\pi_f(\mathrm{If} \circ \mathrm{Jf}):<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu(\pi_f(\mathrm{If}) \circ [1, \pi_f(\mathrm{Jf})]:<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu(\pi_f(\mathrm{If}):<<\pi_f(\mathrm{Ef})>, \mu(\pi_f(\mathrm{Jf}):<<\pi_f(\mathrm{Ef})>, x>)>)$

$= \mu(\pi_f(\mathrm{If}):<<\pi_f(\mathrm{Ef})>, z>)$

$= y$.

Case I5. $\mathrm{Hf} = \alpha(\mathrm{Pf})$, and $\alpha(\mathrm{Pf}):<x_1, \ldots , x_n> \to <y_1, \ldots , y_n>$ in $\leq i+1$ steps. Then

$\mathrm{Pf}{:}x_j \rightarrow y_j$ in $\leq i$ steps, $(1 \leq j \leq n)$,

so that

$$\mu(\pi_f(\mathrm{Pf}){:}<<\pi_f(\mathrm{Ef})>, x_j>) = y_j,\ (1 \leq j \leq n)\ .$$

Therefore, $\mu(\pi_f(\mathrm{Hf}){:}<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu(\pi_f(\alpha(\mathrm{Pf})){:}<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu(\alpha(\pi_f(\mathrm{Pf})) \circ \mathrm{DISTL}{:}<<\pi_f(\mathrm{Ef})>, x>)$

$= \mu(\alpha(\pi_f(\mathrm{Pf})){:}<<<\pi_f(\mathrm{Ef})>, x_1>, \ldots, <<\pi_f(\mathrm{Ef})>, x_n>>)$

$= <\mu(\pi_f(\mathrm{Pf}){:}<<\pi_f(\mathrm{Ef})>, x_1>), \ldots, \mu(\pi_f(\mathrm{Pf}){:}<<\pi_f(\mathrm{Ef})>, x_n>)>$

$= <y_1, \ldots, y_n>$

$= y$. QED

Thus we know that even very complex functions can be represented in the FFP notation. For example, the non-primitive-recursive "Ackermann function" defined by

Def $\mathrm{ack} = \mathrm{eq0} \circ 1 \rightarrow \mathrm{a} \circ 2;\ \mathrm{eq0} \circ 2 \rightarrow \mathrm{ack} \circ [\mathrm{s} \circ 1, \overline{1}];\ \mathrm{ack} \circ [\mathrm{s} \circ 1, \mathrm{ack} \circ [1, \mathrm{s} \circ 2]]$

(where a and s are primitives that, respectively, add and subtract 1) is represented by $<ACK>$ where, according to the algorithm,

ACK

$= \pi_{\mathrm{ack}}(\mathrm{E}(\mathrm{ack}))$

$= \pi_{\mathrm{ack}}(\mathrm{eq0} \circ 1 \rightarrow \mathrm{a} \circ 2;\ \mathrm{eq0} \circ 2 \rightarrow \mathrm{ack} \circ [\mathrm{s} \circ 1, \overline{1}];\ \mathrm{ack} \circ [\mathrm{s} \circ 1, \mathrm{ack} \circ [1, \mathrm{s} \circ 2]])$

$= \pi_{\mathrm{ack}}(\mathrm{eq0} \circ 1) \rightarrow \pi_{\mathrm{ack}}(\mathrm{a} \circ 2);\ \pi_{\mathrm{ack}}(\mathrm{eq0} \circ 2) \rightarrow \pi_{\mathrm{ack}}(\mathrm{ack} \circ [\mathrm{s} \circ 1, \overline{1}]);$
$\quad \pi_{\mathrm{ack}}(\mathrm{ack} \circ [\mathrm{s} \circ 1, \mathrm{ack} \circ [1, \mathrm{s} \circ 2]]))$

$= \mathrm{eq0} \circ 1 \circ 2 \rightarrow \mathrm{a} \circ 2 \circ 2;\ \mathrm{eq0} \circ 2 \circ 2 \rightarrow \pi_{\mathrm{ack}}(\mathrm{ack} \circ [\mathrm{s} \circ 1, \overline{1}]);$
$\quad \pi_{\mathrm{ack}}(\mathrm{ack} \circ [\mathrm{s} \circ 1, \mathrm{ack} \circ [1, \mathrm{s} \circ 2]])$

$= \mathrm{eq0} \circ 1 \circ 2 \rightarrow \mathrm{a} \circ 2 \circ 2;\ \mathrm{eq0} \circ 2 \circ 2 \rightarrow \mathrm{APPLY} \circ [1, [\mathrm{s} \circ 1 \circ 2, \overline{1}]];$
$\quad \mathrm{APPLY} \circ [1, [\mathrm{s} \circ 1 \circ 2, \mathrm{APPLY} \circ [1, [1 \circ 2, \mathrm{s} \circ 2 \circ 2]]]]$

Functions defined by mutually recursive equations

Consider now a collection of functions defined by a set of mutually recursive equations such as

$f = \mathrm{F}fgh$
$g = \mathrm{G}fgh$
$h = \mathrm{H}fgh$

We might try to extend the above approach by iterating the algorithm with respect to the various function letters being defined; *i.e.* we might try letting

$$h' = \pi_h(\mathrm{H}fgh) ,$$

$$\mathrm{G}'fg = \mathrm{G}fgh \text{ with } <h'> \text{ substituted for } h ,$$

$$g' = \pi_g(\mathrm{G}'fg) ,$$

$$\mathrm{F}'f = \mathrm{F}fgh \text{ with } <h'> \text{ substituted for } h \text{ and } <g'> \text{ substituted for } g ,$$

$$f' = \pi_f(\mathrm{F}'f) ,$$

and $<f'>$ = the representation for f.

However, this fails since $\mathrm{G}'fg$ is an object that contains an FFP expression and therefore is not in the domain of definition of π_g, a mapping that takes FP objects as input. Nor can the definition of π_g be extended to include FFP objects. Essentially the difficulty is that determining whether an arbitrary occurrence of an atom A in an FFP object represents a function to be applied or simply the object A is recursively unsolvable.

However, if we modify the algorithm to produce representations with respect to all of the defined functions *simultaneously*, then it is possible to construct correct representations for FP functions defined by mutually recursive equations. The idea is that if $\mathrm{H}fgh$ (some form using the defined functions f, g, and h) is being applied to an object x in FP, then in FFP the object representing $\mathrm{H}fgh$ will be applied to a pair $<ENV, x>$, where ENV is a sequence that contains the representations of all of the functions defined by functional equations. Intuitively, ENV will comprise the "environment" of currently defined functions.

Suppose now that we have a collection of mutually recursive definitions, one of which (say **Def** $f = \mathrm{F}fgh$) defines the function f. We need to choose a representation for f, *i.e.* to define $\pi(f)$, such that the application

$$\pi(f):x$$

will result in the application

$$\pi(\mathrm{F}fgh):<ENV, x>$$

where ENV somehow contains the representations for all of the functions f, g, and h. One way to accomplish this is to let f', g', and h' be the representations of $\mathrm{F}fgh$, $\mathrm{G}fgh$, and $\mathrm{H}fgh$ respectively, and to let

$<f', f', g', h'>$ represent f,
$<g', f', g', h'>$ represent g,
and $<h', f', g', h'>$ represent h.

(We will see that this slightly redundant representation -- having two occurrences of the function being defined -- will make the extension of the representation-producing algorithm simpler and more uniform.)

Note now that

$$\mu(\pi(f):x)$$

$= \mu(<f', f', g', h'>:x)$

$= \mu(f':<<f', f', g', h'>, x>)$

$= \mu(\pi(\mathrm{F}fgh):<<f', f', g', h'>, x>)$

$= \mu(\pi(\mathrm{F}fgh):<ENV, x>)$

where ENV is taken to be the sequence whose $(i+1)^{st}$ element is the representation of the i^{th} defined function and whose first element is any arbitrary FFP object.

All that remains is to develop the extended algorithm so that the meaning of $\pi(\mathrm{F}fgh)$ applied to the pair $<ENV, x>$ in FFP will be the same as the meaning of $\mathrm{F}fgh:x$ in FP. We will use π_D to denote this extended algorithm, since the algorithm will be used to produce representations of FP forms with respect to a set D of defined function symbols.

Algorithm (Extended representation-producing) If D is an ordered set of n atoms denoting defined function names, then the FFP representation of a form E with respect to the set D, $\pi_D(E)$, is constructed recursively by cases on the structure of E:

Form of E	FFP representation of E
p (an atom not in D)	$p \circ 2$
f_i (the i^{th} element of D)	APPLY $\circ$ [[i, 1, 2,..., n] $\circ$ TAIL $\circ$ 1, 2]
$\bar{x}$	$\bar{x}$ if x $\neq \perp$, else UNDEF
$P \rightarrow Q; R$	$\pi_D(P) \rightarrow \pi_D(Q); \pi_D(R)$
$[P, Q]$	$[\pi_D(P), \pi_D(Q)]$
$P \circ Q$	$\pi_D(P) \circ [1, \pi_D(Q)]$
$\alpha(P)$	$\alpha(\pi_D(P)) \circ$ DISTL

(Note that π_D is just the same as π_f except for the case $E = f_i$.)

Then given a set of definitions for the n functions $\mathbf{f} = \{f_1, \ldots, f_n\}$:

Def $f_1 = \mathrm{E}_1\mathbf{f}$
Def $f_2 = \mathrm{E}_2\mathbf{f}$
.
.
.
Def $f_n = \mathrm{E}_n\mathbf{f}$

we will choose for $\pi(f_i)$ (the FFP representation of the function f_i) the $(n+1)$-element sequence

$$<F_i, F_1, F_2, \ldots, F_n>$$

where $F_i = \pi_D(\mathrm{E}_i\mathbf{f})$ for $1 \leq i \leq n$.

To show that these objects correctly represent the defined functions, we need to show for all objects x and FP expressions Hf (*i.e.* functions built up from the primitives and the functions f_i by the application of combining forms) that

$$\mu(\pi_f(\mathrm{Hf}):\langle ENV, x\rangle) = \mathrm{Hf}:x\ .$$

This can be shown by using the proof of the theorem for the single equation algorithm given above with case I1 modified as follows:

Case I1′. $\mathrm{Hf} = f_i$, and $f_i:x \rightarrow y$ in $\leq i+1$ steps. Then

$$\mathrm{E}_i\mathrm{f}:x \rightarrow y \text{ in } \leq i \text{ steps,}$$

so that

$$\mu(\pi_D(\mathrm{E}_i\mathrm{f}):\langle ENV, x\rangle) = y\ .$$

Therefore, $\mu(\pi_D(\mathrm{Hf}):\langle ENV, x\rangle)$

$$= \mu(\pi_D(f_i):\langle ENV, x\rangle)$$

$$= \mu(\pi_D(f_i):\langle\langle Obj, F_1, F_2, \ldots, F_n\rangle, x\rangle) \quad \text{, for some object } Obj$$

$$= \mu(\mathrm{APPLY}\circ[[i, 1, 2,\ldots, n]\circ\mathrm{TAIL}\circ 1, 2]:\langle\langle Obj, F_1, F_2, \ldots, F_n\rangle, x\rangle)$$

$$= \mu(\mathrm{APPLY}:\langle\langle F_i, F_1, F_2, \ldots, F_n\rangle, x\rangle)$$

$$= \mu(\langle F_i, F_1, F_2, \ldots, F_n\rangle:x)$$

$$= \mu(F_i:\langle\langle F_i, F_1, F_2, \ldots, F_n\rangle, x\rangle)$$

$$= \mu(\pi_D(\mathrm{E}_i\mathrm{f}):\langle ENV, x\rangle)$$

$$= y\ .$$

Acknowledgments I am grateful to John Backus, Carl Hauser, Peter Lucas, Steven Muchnick, and Donald Stanat, all of whom read earlier drafts of this paper and made many helpful comments and suggestions. I am particularly indebted to Carl Hauser for suggesting the proof technique used in the main theorem.

References

[Ba 78] Backus, J.W. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *CACM*, August 1978.

[Ch 41] Church, A. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, N.J., 1941.

[Cu 58] Curry, H.B. and Feys, R. *Combinatory Logic, Vol. 1.* North-Holland Pub. Co., Amsterdam, 1958.

[Kl 52] Kleene, S.C. *Introduction to Metamathematics*, Van Nostrand, New York, 1952.

[Mc 60] McCarthy, J. "Recursive functions of symbolic expressions and their computation by machine, Part 1." *CACM*, April 1960.