LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

FP 1:5: Backus' FP with Higher Order Functions

Computation Structures Group Memo 243
January 1982

Arvind
J. Dean Brock
Keshav K. Pingali

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

FP 1.5: Backus' FP with Higher Order Functions
(Summary)

Arvind, MIT
J. Dean Brock, UNC-CH
Keshav K. Pingali, MIT

1. Introduction

Programming language design can be regarded as a search for useful constraints that guide a programmer in the writing of correct programs. From this point of view, extending any small, elegant language would appear to be an exercise in futility, especially in the absence of a strong practical need to do so. In spite of this, we would like to carry out this exercise with Backus' FP language [2]. We would like to extend FP to permit user-defined higher order functions. The importance of higher order functions is often not appreciated because of a lack of experience in programming with them. However, the few examples of programming with higher order functions that we have seen in the literature [3, 5] tend to support our viewpoint. It seems to us that there should be a place for higher order functions in a programming language.

Extensions have a price. At the minimum, the language gets bigger, but more often, some property of the language gets destroyed inadvertently (*e g.* pure LISP to LISP1.5). Introducing higher order functions in FP requires including, at the very least, the `apply` primitive. (The meaning of `apply` $:< f, a >$ is $f : a$.) There are some technical difficulties in doing so (see page 632 in [2]): Backus allows `apply` and user-defined higher order functions in FFP, but that makes FFP non-extensional. We will show that FP1.5 does have extensionality. It appears that the only cost of introducing higher order functions is the complexity of the semantic domain of FP1.5 in comparison with the semantic domain of FP. Undeniably, the most useful property of FP is the algebraic identities whose variables range over all FP programs. We will prove that all the algebraic identities given by Backus [2] for FP programs hold for all FP1.5 programs.

2. The Language

An FP1.5 system is comprised of a set O of objects, and a set E of expressions. The set O has *simple atoms* like `true`, `false`, the integers, <> (the empty sequence), etc. In addition, it includes *function objects.* Function atoms are the usual functions like +, −, `S1`, `S2`... (where `S`n is function that returns the n'th element of a sequence) and `apply`, as well as four *program-building operators* called `formcomp`, `formkonst`, `formnf`, and `formcond`. A program-building operator takes a sequence of function atoms or objects, and returns a function object - for example, `formcond` takes a sequence of three function objects (say $< p, f, g >$) and returns the function object $(p \rightarrow f; g)$; this function object when applied to an object x, returns $f(x)$ if $p(x)$ is true, and returns $g(x)$ if $p(x)$ is false. The other program-building operators are similar. A sequence of objects is also an object (but not a function object).

No operator is allowed to "look inside" any function objects to which it is applied. This rule is a *must* for extensionality. One of its consequences is that there can be no function which tests for the equality of two function atoms or objects.

An FP1.5 expression is either an FP1.5 object, or a function object applied to an object. We will now give the syntax of the language.

$< expression >::=< object > | < funobject > : < object >$

$< object >::=< sequence > | < simatom > | < funobject > | \perp$

$< sequence >::=<< object >, < object >, \ldots >$

$< simatom >::= \texttt{true}, \texttt{false}, 0, 1, 2, \ldots$

$< funobject >::= < funatom > | < composition > | < conditional > | < constructor >$
$| < konstant >$

$< funatom >::= +, -, \ldots, \texttt{S1}, \texttt{S2}, \ldots, \texttt{apply}, \texttt{formkonst}, \texttt{formcomp}, \texttt{formnf}, \texttt{formcond}$

$< composition >::=< funobject > \circ < funobject > \circ \ldots < funobject >$

$< conditional >::= (< funobject > \rightarrow < funobject >; < funobject >)$

$< constructor >::= [< funobject >, \ldots < funobject >]$

$< konstant >::= \overline{< object >}$

Like FP, we allow a set of function definitions (called the D set). A D set is a sequence of definitions of the form:

def $< funatom >=< funobject >$

which associates the function atoms of the D set with function objects. The function δ maps each function atom into its function object definition.

We will now define the meaning function μ for FP1.5. μ is defined with reference to two functions: ρ, which associates function atoms with their function representations, and δ, which associates atoms of the D-set with their function objects. The domains of these three functions are:

$\mu :< expression > \rightarrow < object >$

$\rho :< funatom > \rightarrow [< object > \rightarrow < object >]$

$\delta :< funatom > \rightarrow < funobject >$

The abstract interpreter for these functions is given below. Though not explicitly stated, it is assumed that function applications in the abstract interpreter are strict or error preserving.

$\mu\ E = E$, if E is an $< object >$

$\mu\ F : E = \mu\ (\delta F : E)$, if F an atom in the D-set

$= \rho F(E)$, if F is a $< funatom >$

$= \perp$, otherwise

$\mu\ \overline{F} : E = F$

$\mu\ F \circ G : E = \mu\ F : (\mu\ G : E)$

$\mu\ [F_1, ..., F_n] : E =< \mu\ F_1 : E, ..., \mu\ F_n : E >$

$$\mu\,(P \rightarrow F;G) : E = \mu\, F : E, \quad \text{if } \mu\, P : E = \texttt{true}$$
$$= \mu\, G : E, \quad \text{if } \mu\, P : E = \texttt{false}$$
$$= \perp, \quad \text{otherwise}$$

Finally, the function representation of our five new program-building function atoms may be defined as:

ρ apply $(< F, X >) = \mu\, F : X$, if F a $< \textit{funobject} >$ and X an $< \textit{object} >$, otherwise $\perp$

ρ formkonst$(X) = \overline{X}$

ρ formcomp$(< F, G >) = F \circ G$

ρ formnf$(< F_1, ..., F_n >) = [F_1, ...F_n]$

ρ formcond$(< P, F, G >) = (P \rightarrow F; G)$

Note, we have adopted a non-denotational view of the program-building atoms. We do not consider an application of these atoms to yield an abstract function, *but* to yield the FP syntax for a function.

Theorem: FP1.5 is extensional – that is, if $X \equiv Y$, then for all function objects H, $\mu\, H : X \equiv \mu\, H : Y$. Intuitively, the idea is that there exists an (interesting) equality predicate on programs such that if two objects are equal under this predicate, then there is no function in the language that can distinguish between them. The equality predicate that we are considering is defined so that $X \equiv Y$ if and only if (1) X and Y are simple atoms and $X = Y$; (2) X and Y are functions and for all objects E, $\mu\, X : E \equiv \mu\, Y : E$; and (3) X and Y are sequences of the length n, where $X =< X_1, \ldots X_n >$ and $Y =< Y_1, ...Y_n >$, and $X_i \equiv Y_i$ for all i from 1 to n.

Scott's [4] semantic theory assures us of the existence of such an equality predicate.

Proof of extensionality:

We will prove that our function is extensional by means of an induction on the interpretation, by μ, of expressions in FP1.5.

Case 1 – It is easy to verify that if H is a $< funatom >$ such as +, −, S1, etc., $\mu\ H : X \equiv \mu\ H : Y$. We will show this for S1. If X is not a sequence, then neither is Y, and hence, $\mu\ H : X$ and $\mu\ H : Y$ are both $\perp$. If X is a sequence, then so is Y, and they must both be either the empty sequence, in which case μ S1 $: X = \perp = \mu$ S1 $: Y$, or they are both non-empty, in which case $X =< X_1, \ldots >$ and $Y =< Y_1, \ldots >$, and μ S1 $: X = X_1 \equiv Y_1 = \mu$ S1 $: Y$.

Case 2 – If H is `formkonst`, then $\mu\ H : X$ is $\overline{X}$ which equals $\overline{Y}$, as for all Z, $\mu\ \overline{X} : Z = X \equiv Y = \mu\ \overline{Y} : Z$. If H is `formcomp`, then if X is not a sequence, neither is Y and $\mu H : X = \perp = \mu H : Y$. If the sequence X contains objects that are not function objects, then again $\mu\ H : X = \perp = \mu\ H : Y$. Finally, without loss of generality assume that X is a sequence $< FX, GX >$ of function objects FX and GX. Then Y is an object $< FY, GY >$, and $FX \equiv FY$ and $GX \equiv GY$. Consequently, $\mu\ H : X = FX \circ GX$ and $\mu\ H : Y = FY \circ GY$, and since $\mu\ FX \circ GX : E = \mu\ FX : (\mu\ GX : E) \equiv \mu\ FY : (\mu\ GY : E) = \mu\ FY \circ GY : E$ for all E, $\mu\ H : X \equiv \mu\ H : Y$. The proofs for the remaining program-building operators are similar.

Case 3 – If H is apply, then let us consider only the error-free case where H is applied to the sequences $< FX, X >$ and $< FY, Y >$, where FX and FY are function objects and $FX \equiv FY$ and $X \equiv Y$. Then $\mu\ H : < FX, X >= \mu\ FX : X$ and $\mu\ H : < FY, Y >= \mu\ FY : Y$. Because $FX \equiv FY$ by our definition of function equality $\mu\ FX : X \equiv \mu\ FY : X$, and because $X \equiv Y$ by our induction hypothesis, $\mu\ FY : X \equiv \mu\ FY : Y$. Therefore, $\mu\ H : < FX, X > \equiv \mu\ H : < FY, Y >$.

Case 4 – The cases for the conditional, composition, constructor and constant operators are straightforward.

We have given an outline of a proof of extensionality above - the complete proof will be given in the paper.

It is easy to check that the algebraic laws given by Backus for FP also hold for FP1.5. Two of these laws are given below:

$$\begin{aligned}
\mu\,[F1, F2] \circ\ G : E &= \mu\,[F1, F2] : (\mu\, G : E) \\
&= < \mu\, F1 : (\mu\, G : E), \mu\, F2 : (\mu\, G : E) > \\
&= < \mu\, F1 \circ\ G : E, \mu\, F2 \circ\ G : E > \\
&= \mu\,[F1 \circ\ G, F2 \circ\ G] : E
\end{aligned}$$

$$\begin{aligned}
\mu H \circ (P \rightarrow\ F; G) : E &= \mu\, H : (\mu(P \rightarrow F; G) : E) \\
&= \mu\, H : (\mu\, F : E), \quad \text{if } \mu\, P : E \text{ is } \texttt{true} \\
&= \mu\, H \circ F : E \\
&= \mu\, (P \rightarrow\ H \circ\ F; H \circ\ G) : E
\end{aligned}$$

The case where $\mu\ P : E$ is `false` is similar.

In a similar fashion, it is easy to verify that all the relevant laws given by Backus in [2] are valid in our system.

3. Examples

In his use of streams, Burge [3] has shown that expressions with higher order functions can be used to specify "infinite" computation that can be evaluated in a conventional, non-lazy manner. This same technique can be used in FP1.5 by representing a stream (list) as a constant function which maps any non-$\perp$ value to a pair consisting of the first element of the stream, a simple value, and the remainder of the stream, itself a stream.

Following this representation, the FP1.5 expression which we call *FiveForever*, the infinite sequence of fives, must be defined so that:

$\mu\ FiveForever : X =< 5, FiveForever >$

One FP1.5 expression having this property may be written as:

def $FiveForever = [\overline{5}, \overline{FiveForever}]$

One can generalize this to an expression *Forever* which, when applied to a value, yields the infinite stream consisting only of that value. This stream-generating function may be defined as:

def $Forever =$ **formnf** $\circ$ [**formkonst**,
formcomp $\circ$ [$\overline{\textbf{apply}}$,
formnf $\circ$ [**formkonst** $\circ$ $\overline{Forever}$,
formkonst]]]

An application of μ to this admittedly ugly expression (which will soon be beautified through the application of syntactic sugar) will illustrate that *Forever* does fulfill its intended role, for:

$$\mu\ Forever : 5 = [\overline{5}, \textbf{apply} \circ [\overline{Forever}, \overline{5}]]$$

and when this stream result is applied to any non-$\perp$ argument we see that:

$$\mu\ (\mu\ Forever : 5) : X =< 5, \mu\ Forever : 5 >$$
$$=< 5, [\overline{5}, \textbf{apply} \circ [\overline{Forever}, \overline{5}]] >$$

Note how functionals have been used to "stop" the evaluation of the infinite stream after the production of its first element. With this stream representation the operators *first* (*car*), *rest* (*cdr*), and *cons* may be easily defined as:

def *first* = **S1** $\circ$ **apply** [**id**, $\overline{0}$]

def *rest* = **S2** $\circ$ **apply** [**id**, $\overline{0}$]

def *cons* = **formkonst**

The previously-promised syntactic sugar can greatly improve the readability of the example FP1.5 expressions. We will use the double circle $\odot$ as a sweetener for `formcomp` by allowing

$$F_1 \odot \ldots \odot F_n$$

to stand for

$$\texttt{formcomp} \circ [F_1, \ldots F_n].$$

We will also allow $[\![$, $]\!]$, and $\Rightarrow$ to be used in the following abbreviations:

$$\begin{aligned} [\![F_1, \ldots, F_n]\!] &= \texttt{formnf} \circ [F_1, \ldots, F_n] \\ P \Rightarrow F, G &= \texttt{formcond} \circ [P, F, G] \end{aligned}$$

Using these sugarings, *Forever* may be more tastefully defined as:

$$\texttt{def}\quad \mathit{Forever} = [\![\, \texttt{formkonst}, \overline{\texttt{apply}} \odot [\![\, \overline{\overline{\mathit{Forever}}}, \texttt{formkonst} \,]\!] \,]\!]$$

Many FP functional forms such as α (apply to all) can be easily defined using the extended FP1.5 syntax. For example:

$$\begin{aligned} \texttt{def}\ \mathit{AA} = \overline{\texttt{eq0}} \odot \overline{\texttt{ln}} \Rightarrow\ & \\ & \overline{[]}, \\ & \overline{\texttt{appndl}} \odot [\![\, \texttt{id} \odot \overline{\texttt{s1}}, \\ & \qquad \overline{\texttt{apply}} \odot [\![\, \overline{\mathit{AA}} \odot \texttt{formkonst}, \overline{\texttt{tl}} \,]\!] \,]\!] \end{aligned}$$

Note again, the use of $\overline{\texttt{apply}}$ to control a potentially unbounded recursion.

4. FP1.5 and FFP

We illustrate some differences between FFP and FP1.5 using a simple example. Consider the higher order function *twice* given in [3].

def *twice* $f \quad x = f(fx)$

Of course, twice cannot be written in FP. In FP1.5 it may be written as follows:

def *twice* = **apply** $\circ$ [**S1**, **apply**]

where we have assumed that the argument is of the form $< f, x >$. This definition of twice can be directly translated in FFP also. However, one could also define a new functional form $<$ *twice*, $f >$ which applies f two times. In that case twice would be the FFP expression corresponding to the following:

apply $\circ$ [2 $\circ$ 1, **apply** $\circ$ [2 $\circ$ 1, 2]]

This later form cannot be part of FP1.5 because sequences in FP1.5 do not represent functions (because there is no meta composition rule).

Extensionality is not destroyed by the inclusion of **apply** or function-forming operations – it can only be destroyed by operations that tear functions apart. In FFP this power is provided by permitting the representation of a function to be a sequence which can be taken apart by the selector functions. FP1.5 forbids this.

References

[1] Backus, J., "Programming Language Semantics and Closed Applicative Language," *Proceedings of the ACM Symposium on Principles of Programming Languages*, October 1973, 71-86.

[2] Backus, J., "Can Programming Be Liberated from the von Newmann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* **21**, 8(August 1978), 613-641.

[3] Burge, W. H., *Recursive Programming Techniques*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1975.

[4] Scott, D. S., "Data Types as Lattices," *SIAM Journal of Computing* **5**, 3(September 1976), 522-587.

[5] Turner, D. A., "The Semantic Elegance of Applicative Languages," *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, October 1081, 85-92.