A Note on the Under-Appreciated For-Loop

José N. Oliveira
Abstract

This short research report records some thoughts concerning a simple algebraic theory for for-loops arising from my teaching of the Algebra of Programming course at the University of Minho. Interest in this so neglected recursion-algebraic combinator grew recently after reading Olivier Danvy’s paper on folding over the natural numbers. The report casts Danvy’s results as special case of the powerful adjoint-fold theorems of the Algebra of Programming.
A Note on the Under-Appreciated For-Loop

José N. Oliveira

Oct 2020

Abstract

This short research report records some thoughts concerning a simple algebraic theory for for-loops arising from my teaching of the Algebra of Programming course at the University of Minho. Interest in this so neglected recursion-algebraic combinator grew recently after reading Olivier Danvy’s paper on folding over the natural numbers. The report casts Danvy’s results as special case of the powerful adjoint-fold theorems of the Algebra of Programming.

1 Context

I have been teaching the Algebra of Programming in a 2nd year course at Minho University since academic year 1998/99, just a few days after AFP’98 took place in Braga, where my department is located. Since then experience has been gathered on how to teach this topic [5], which is regarded as inaccessible by many people.

Quite often I deviate from common practice in my lectures. For instance, my first example of monadic programming is not a stateful computation or a I/O primitive. Instead, I rather use the finite distribution monad [2] to throw two dice on a Monopoly board with just one single line of code:

> do { a <- uniform [1..6]; b <- uniform [1..6]; return(a+b) }

7 16.7%
6 13.9%
8 13.9%
5 11.1%
9 11.1%
4 8.3%
10 8.3%
3 5.6%
11 5.6%
2 2.8%
12 2.8%

By challenging the students to write equivalent code in their favourite languages, which as a rule are C++, Java or Python, I intend to show how expressive the device is (the monad) that they are going to learn. However, I never have time to show how monads arise from adjunctions, which is a pity. (As section 3 below shows, I believe.)
Another deviation from common practice is to use for-loops to introduce fold/unfold combinators (and catamorphisms, anamorphisms and so on) instead of the corresponding operations on lists. Just remove two letters from the word “foldr” and you get “for”; just remove the multiplicand $A \times$ from the finite list recursion pattern $1 + A \times X$ and you get $1 + X$, the recursive pattern of Peano natural numbers (instead of lists).

This saves the students from a (perhaps too early) explanation of the parameterization on $A$ and has the interesting advantage of bridging directly with school maths through instances of a scheme easy to apprehend [5]. Take for instance the following properties of multiplication, which can be found in any K7-K8 maths textbook, and restrict them to the natural numbers:

$$\begin{align*}
a \times 0 &= 0 \\
a \times 1 &= a \\
a \times (b + c) &= a \times b + a \times c
\end{align*}$$

From these the two line program

$$\begin{align*}
a \times 0 &= 0 \\
a \times (b + 1) &= a \times b + a
\end{align*}$$

arises by making $c := 1$ in the distributivity property and simplifying (whereby the second property disappears because it is entailed by the other two). Then single out section $(a \times)$,

$$\begin{align*}
(a \times) 0 &= 0 \\
(a \times) (b + 1) &= (a +) ((a \times) b)
\end{align*}$$

and go pointfree ($\mathbf{1}_k$ denotes the everywhere $k$ constant function and $\text{succ } b = b + 1$):

$$\begin{align*}
(a \times) \cdot 0 &= 0 \\
(a \times) \cdot \text{succ} &= (a +) \cdot (a \times)
\end{align*}$$

This immediately yields the $(1 +)$-catamorphism

$$(a \times) = \mathbf{1} [[0, a +]]$$

over the natural numbers, which can also be written as

$$(a \times) = \text{for } (a +) \cdot 0$$

by defining the for-combinator:

$$\text{for } s \; z = [[z, s]]$$

Thus for $s \; z \; n$ means to iterate the loop body $s \; n$-times starting with initial value $z$. And $(a \times)$ is just $(a +)$ iterated.

Framed in the wider setting of initial algebra controlled recursion, the for $s \; z$ combinator inherits a number of properties stemming from initiality. For instance, the calculation of the following property,

$$s \cdot (\text{for } s \; z) = \text{for } s \; (s \; z)$$

This immediately yields the $(1 +)$-catamorphism

$$(a \times) = \mathbf{1} [[0, a +]]$$

over the natural numbers, which can also be written as

$$(a \times) = \text{for } (a +) \cdot 0$$

by defining the for-combinator:

$$\text{for } s \; z = [[z, s]]$$

Thus for $s \; z \; n$ means to iterate the loop body $s \; n$-times starting with initial value $z$. And $(a \times)$ is just $(a +)$ iterated.
resorts to “cata-fusion” [5]:

\[ s \cdot (\text{for } s \ z) = \text{for } s (s \ z) \]

\[ \Leftrightarrow \]

\[ \{ \text{(5) twice } \} \]

\[ s \cdot (\{ z, s \}) = (\{ s \cdot z, s \}) \]

\[ \Leftarrow \]

\[ \{ \text{cata-fusion } \} \]

\[ s \cdot [z, s] = [s \cdot z, s \cdot s] \]

\[ \Leftrightarrow \]

\[ \{ \text{coproducts ; constant functions } \} \]

\[ [s \cdot z, s \cdot s] = [s \cdot z, s \cdot s] \]

\[ \square \]

2 Calculating for-loops

Although the (iterative) functions that one can write with the \texttt{for}-combinator alone are not very interesting, they become far more so once we exploit the theory of catamorphisms, for instance the mutual-recursion law:

\[ \langle f, g \rangle = \L (\langle h, k \rangle) \Leftrightarrow \begin{cases} f \cdot \text{in} = h \cdot F \langle f, g \rangle \\ g \cdot \text{in} = k \cdot F \langle f, g \rangle \end{cases} \] (7)

In the case of natural numbers, \texttt{in} = \{0, succ\} is the Peano algebra of natural numbers, for \( F X = 1 + X \).

The law extends to more than two mutually (primitive) recursive functions. Therefore, once one is able to express the function of interest in a system of mutually (1+) -recursive functions\(^1\), its implementation as a for-loop is immediately derivable by (7). Take factorial, for instance:

\[ \{ \text{fac 0 = 0} \}
\]
\[ \{ \text{fac (n + 1) = (n + 1) * fac n} \}
\]

As \( n + 1 = \text{for } (1+) 1 \), term \( n + 1 \) can be thought of as in mutual recursion with \texttt{fac}. By (7) one gets iterative factorial:

\[ \text{fac n} = m \text{ where}
\]
\[ (\_, m) = \text{for loop } (1, 1) n
\]
\[ \text{loop } (s, n) = (1 + s, s * n) \]

My notes [5] have several examples of this programming technique, including the derivation of for-loop implementations of \( \mathbb{R} \)-valued functions approximated by their corresponding Taylor series. For instance, the exponential function \( e^x : \mathbb{R} \to \mathbb{R} \) has Taylor series:

\[ e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \] (8)

\(^1\)That is, primitive recursion functions.
The three threads of computation in this formula are made explicit in the following system of mutually recursive functions, where \( \exp x n \) computes the \( n \)-step approximation of \( e^x \):

\[
\begin{align*}
\exp x 0 &= 1 \\
\exp x (n + 1) &= h x n + \exp x n \\
h x 0 &= x \\
h x (n + 1) &= \frac{x}{2\pi n} (h x n) \\
s2 0 &= 2 \\
s2 (n + 1) &= 1 + s2 n
\end{align*}
\]

By (7) one calculates the following for-loop implementation

\[
\text{exp } x n = e \quad \text{where} \\
(e, \ldots, \ldots) = \text{for loop init } n \\
\text{init } = (1, 2, x) \\
\text{loop } (e, s, h) = (h + e, s + 1, (x / s) * h)
\]

which could be translated directly to the encoding in e.g. C, if needed:

```c
float exp(float x, int n)
{
    float h=x; float e=1; int s=2; int i;
    for (i=0;i<n+1;i++) {e=e+h;h=(x/s)*h;s++;}
    return e;
}
```

Functions involving more calculation threads require larger mutual recursion bundles, for instance

\[
\cos x = \sum_{i=0}^{\infty} \frac{(-1)^i (2i)!}{(2^i)!} x^{2i}
\]

which leads to a for-loop involving four variables:

\[
\cos x = \text{prj} \cdot \text{for loop init where} \\
\text{loop } (c, f, j, k) = (c + f, (-x \uparrow 2 / j) \ast f, j + k, k + 8) \\
\text{init } = (1, -x \uparrow 2 / 2, 12, 18) \\
\text{prj } (c, \ldots, \ldots) = c
\]

In a sense, the mutual recursion law (7) gives us a hint on how many global variables are needed and how they “are born” in computer programs, out of the maths definitions themselves.

### 3 Adjoint for-loops

It is well known that, in the case of lists, \texttt{foldr} comes with its “left”-version \texttt{foldl}, which has the same functionality and implements \texttt{foldr} more efficiently under some mild conditions. What about \texttt{for}? Danvy [1] shows that a “left”-version of \texttt{for} exists and that the right and left versions of the combinator are equivalent. Proofs are discharged by natural number induction.
In this section I would like to show that such tail-recursive, “left-for” iterator does not arise by sheer invention but rather from the adjunction that underlies the continuation monad, as instance of a very general device known as adjoint-folds [3].

The machinery may sound too heavy compared to natural number induction but it has the pro of framing [1] in a theory of much wider scope.

**Iteration: for-loops and forl-loops**  
By expanding definition for $s z = (\llbracket z, s \rrbracket)$ (5) to the pointwise level we get:

\[
\begin{cases}
  \text{for } s z 0 = z \\
  \text{for } s z (n + 1) = s \text{ (for } s z n) \\
\end{cases} 
\]  
(9)

The corresponding “left”-version given in [1] is (with the notation and function terminology adapted)

\[
\begin{cases}
  \text{forl } s z 0 = z \\
  \text{forl } s z (n + 1) = \text{forl } s (s z) n \\
\end{cases} 
\]  
(10)

Clearly, for $s z$ is a (1+)-catamorphism while forl $s z$ is not — it is **tail recursive**. How does one prove that they are equivalent using algebra of programming techniques? Note that, by flipping forl $s$, we get

\[
\begin{align*}
\text{forl } s 0 z &= z \\
\text{forl } s (n + 1) z &= \text{forl } s n (s z)
\end{align*}
\]

which points to a (1+)-pattern of recursion but at a higher order, cf.

\[
\begin{align*}
\text{forl } s 0 &= id \\
\text{forl } s (n + 1) &= \lambda z \rightarrow \text{forl } s n (s z)
\end{align*}
\]

Before completing the exercise, one asks: why **flipping**? why a higher-order for-loop?

The answer below relates the tail-recursion pattern of the explicit definition (10) and the Peano (1+)-recursion of (9) via adjoint-folds [3], in particular via the adjoint-catamorphism theorem given below.

**Contravariant exponential functor**  
Let $F : X = K^X$ for some non empty $K$.

Functor $F$ is contravariant: given \[ A \xrightarrow{f} B \], $K^A \xleftarrow{K^f} K^B$ is defined by $K^f = (f)$.

**Contravariant exponential adjunction**  
It turns out that $F$ is adjoint of itself on the opposite category, meaning that we have to reverse the arrows on the right handside of the usual diagram:

\[
\begin{tikzcd}
  k = \text{flip } f \\
  A \xrightarrow{\beta_{\text{ip}} k} K^B & K^{(K^B)} \xleftarrow{\epsilon} B
\end{tikzcd}
\]  
(11)
The induced monad is the continuity monad, \( M X = K^{(K^X)} \). The counit \( \epsilon \) encodes the continuation style: given \( b \in B \) and a continuation \( f \), then \( \epsilon b f = f b \).

In the diagram, \( f b = ((k) \cdot \epsilon) b = (\epsilon b) \cdot k \) and thus \( f b a = \epsilon b (k a) = k a b \). That is: \( f = \text{flip } k \). So isomorphism \( \text{flip} \) is its own inverse.

From this we get the standard properties of adjunctions tuned to (11): reflection,

\[ \text{flip } \epsilon = \text{id} \]  

(12)
cancellation,

\[ (\cdot \text{flip } f) \cdot \epsilon = f \]  

(13)
fusion,

\[ (\text{flip } h) \cdot g = \text{flip } (K^g \cdot h) \]  

(14)
absorption,

\[ K^g \cdot \text{flip } h = \text{flip } (h \cdot g) \]  

(15)
and functor definition:

\[ K^h = \text{flip } (\epsilon \cdot h) = (\cdot h) \]  

(16)

Also of interest is the relation between the constant function combinator \( k \) and the identity:

\[ K \cdot \text{id} \rightarrow K^{K^1} = \text{flip } (1 \overset{\text{id}}{\rightarrow} K^K) \]  

(17)

**Adjoint catamorphism theorem** (Proof in [5].) Let adjunction \( L \dashv R \) be given, with the choice of symbols clearly indicating which functor is the lower adjoint (L) and which is the upper adjoint (R):

\[ L A \rightarrow B \overset{\cong}{\rightarrow} A \rightarrow R B \]  

(18)

Let \( \text{T} \overset{\text{in}}{\rightarrow} \text{F} \text{T} \) be an inductive type and \( \phi : L F \rightarrow G L \) be a natural transformation, that is, free theorem

\[ \phi \cdot (L F k) = (G L k) \cdot \phi \]  

(19)
holds, for some functor \( G \). Then:

\[ f \cdot (L \text{ in}) = h \cdot G f \cdot \phi \iff [f] = [h \cdot G \epsilon \cdot \phi] \]  

(20)

In words, the theorem shows how to convert a \( G \)-hylomorphism \( f \) into its adjoint \( F \)-catamorphism \([f] \) across the adjunction \( L \dashv R \). Expressed by diagrams, it becomes
clear that the lefthand side of (20) is the $G$-hylomorphism

\[
\begin{array}{c}
L \mu_F \\
\downarrow \phi \\
A \leftarrow h \\
\end{array}
\quad \quad \quad
\begin{array}{c}
G L \mu_F \\
\downarrow G_f \\
G A \\
\end{array}
\]

which is converted into its adjoint $F$-catamorphism:

\[
\begin{array}{c}
\mu_F \\
\downarrow \phi \\
\begin{array}{c}
[f] \\[h \cdot G \epsilon \cdot \phi]
\end{array} \\
\begin{array}{c}
\begin{array}{c}
R A \\
\downarrow h \\
G A
\end{array} \\
\begin{array}{c}
\begin{array}{c}
G \epsilon \\
\downarrow L \epsilon \\
\begin{array}{c}
\begin{array}{c}
L F R A \\
\downarrow \phi \\
L F R A
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\]

\[
[f] = \langle h \cdot (G \epsilon \cdot \phi) \rangle
\]

**Peano adjoint catamorphisms** Let us show that for $s$ is one such hylomorphism, for $h = \langle id, s \rangle$ in (20). Since $K^{1+X}$ is isomorphic to $K \times K^X$, it is easy to find the pattern of recursion $G X$ that is adjoint to the Peano one, $F X = 1 + X$:

\[
\begin{align*}
F X &= 1 + X \\
G X &= K \times X \\
\phi : K^F X &\to G (K^X)
\end{align*}
\]

That is, $\phi : K^{1+X} \to K \times K^X$ is the natural isomorphism bridging $(1+)$-recursion and $(K \times)$ recursion. Its converse $\phi^\circ : K \times K^X \to K^{1+X}$ is the function:

\[
\phi^\circ (k, f) = [k, f]
\] (21)

The flip of $\phi^\circ$ will be needed below. Let us calculate it:

\[
1 + X \xrightarrow{\text{flip} \ \phi^\circ} K^{K \times K^X} = [f, g]
\]

yields $f = \pi_1$ by (15,17). Concerning $g$, we see that $\text{flip} \ g$ has type $K \times K^X \to K^X$. So $g = \text{flip} \ \pi_2$. All together:

\[
\text{flip} \ \phi^\circ = [\pi_1, \text{flip} \ \pi_2]
\] (22)
The diagrams below of our instance of the generic theorem already show the arrows reversed due to the contravariance: any function \( f \) of recursive pattern \( K \times K^0 \rightarrow K^{1+N_0} \) converts into its adjoint \( F \)-catamorphism:

Let \( K = A \) and \( h = \langle id, s \rangle \) for some \( s : A \rightarrow A \) in the diagram of \( f \) above. Then hylo \( f \) unfolds as follows:

\[
\begin{align*}
\text{forl } s & = f \\
\{ \text{ introduce forl } s = f \text{ to make } f \text{ parametric on } s \}
\end{align*}
\]

Thus we reach forl \( s \) for the particular choice of \( h = \langle id, s \rangle \). Next, let us calculate its adjoint, which is the Peano catamorphism:

\[
\text{flip (forl } s \) = \langle \text{flip (} \phi^o \cdot (id \times \epsilon) \cdot \langle id, s \rangle \rangle
\]
We start by simplifying \( \text{flip} \ (\phi \circ (id \times \epsilon) \cdot (id, s)) \):

\[
\text{flip} \ (\phi \circ (id \times \epsilon) \cdot (id, s))
\]

\[
= \quad \{ \text{products; absorption (15)} \}
\]

\[
K^{(id, \epsilon \cdot s)} \cdot \text{flip} \ \phi
\]

\[
= \quad \{ (22) \}
\]

\[
K^{(id, \epsilon \cdot s)} \cdot [\pi_1, \text{flip} \ \pi_2]
\]

\[
= \quad \{ \text{coproducts; constant functions; (22)} \}
\]

\[
[\underbrace{K^{(id, \epsilon \cdot s)} \cdot \pi_1}, \text{flip} \ (\pi_2 \cdot (id, \epsilon \cdot s))]
\]

\[
= \quad \{ (16); \text{products} \}
\]

\[
[id, \text{flip} \ (\epsilon \cdot s)]
\]

\[
= \quad \{ \text{exponential functor (16)} \}
\]

\[
[id, (\cdot s)]
\]

Thus:

\[
\text{flip} \ (\text{forl} s) = ([id, (\cdot s)]) = \text{for} (\cdot sl \ id)
\]  

\[\text{(24)}\]

**Forl equal to for.** We finally prove the equality

\[
\text{forl} = \text{for}
\]

holds:

\[
\text{forl} s = \text{forl} s
\]

\[
\Leftrightarrow \quad \{ \text{isomorphism } \text{flip}; (24) \}
\]

\[
\text{flip} \ (\text{forl} s) = ([id, (\cdot s)])
\]

\[
\Leftrightarrow \quad \{ \text{cata-universal} \}
\]

\[
\left\{ \begin{array}{l}
\text{flip} \ (\text{forl} s) \cdot 0 = \text{id} \\
\text{flip} \ (\text{forl} s) \cdot \text{succ} = \text{flip} \ (\text{forl} s \cdot s)
\end{array} \right.
\]

\[
\Leftrightarrow \quad \{ (17); \text{drop isomorphism } \text{flip} \}
\]

\[
\left\{ \begin{array}{l}
(\cdot 0) \cdot \text{forl} s = \text{forl} s \\
(\cdot \text{succ}) \cdot \text{forl} s = \text{forl} s \cdot s
\end{array} \right.
\]

\[
\Leftrightarrow \quad \{ \text{apply all sides to } z \text{ and simplify} \}
\]

\[
\left\{ \begin{array}{l}
\text{forl} s \cdot 0 = \text{forl z} \\
\text{forl} s \cdot \text{succ} = \text{forl} (s \ z)
\end{array} \right.
\]

\[
\Leftrightarrow \quad \{ (6); \text{cata-universal} \}
\]

\[
\text{forl} z = ([z, s])
\]

\[
\Leftrightarrow \quad \{ \text{cata-universal} \}
\]

\[
\text{true}
\]
4 Summary

The first part of this note intends to draw attention to the potential of programming with a combinator as elementary as the for-loop. Danvy [1] gives some examples of its usefulness, to which we had some from our Algebra of Programming classes and lab assignments. The second part shows how tail-recursion arises intrinsically linked to primitive recursion via contravariant exponential adjunction. That is, Peano-recursion is adjoint to tail-recursion.

The calculation of for-loops from mutually primitive recursion functions has, in practice, a worker-wrapper flavour: the wrapper is just a projection that discards the variables that are not of interest to the final result. But the mutually recursive scheme shows that the other variables also “make sense” and calculate useful functions, which are discarded.

Not discarding them could be a way of making non-injective functions available inside “reversible envelopes” needed in e.g. quantum computing, since function pairing can only increase injectivity [4].

References


²URL: http://www.di.uminho.pt/ jno/ps/pdbc.pdf