A Note on the Under-Appreciated For-Loop

José N. Oliveira

Techn. Report TR-HASLab:01:2020

Oct 2020

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:01:2020
A Note on the Under-Appreciated For-Loop
by 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 to 2nd year courses 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 cases of the powerful adjoint-catamorphism theorem 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 to 2nd year courses 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 cases of the powerful adjoint-catamorphism theorem of the Algebra of Programming.

1 Context

I have been teaching Algebra of Programming to 2nd year courses at Minho University since academic year 1998/99, starting 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 colleagues, let alone software practicioners.

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:

$$\left\{\begin{array}{l} a \times 0 = 0 \\ a \times 1 = a \\ a \times (b + c) = a \times b + a \times c \end{array}\right. \tag{1}$$

From these the two line program

$$\left\{\begin{array}{l} a \times 0 = 0 \\ a \times (b + 1) = a \times b + a \end{array}\right. \tag{2}$$

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 $(a\times)$,

$$\left\{\begin{array}{l} (a\times)\ 0 = 0 \\ (a\times)\ (b + 1) = (a+)\ ((a\times)\ b) \end{array}\right. \tag{3}$$

and go *pointfree* ($\underline{k}$ denotes the everywhere k constant function and $\mathsf{succ}\ b = b + 1$):

$$\left\{\begin{array}{l} (a\times) \cdot \underline{0} = \underline{0} \\ (a\times) \cdot \mathsf{succ} = (a+) \cdot (a\times) \end{array}\right.$$

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

$$(a\times) = (\!|\, [\underline{0}, a+] \,|\!)$$

over the natural numbers, which can also be written as

$$(a\times) = \mathsf{for}\ (a+)\ 0 \tag{4}$$

by defining the **for**-combinator:

$$\mathsf{for}\ s\ z = (\!|\, [\underline{z}, s] \,|\!) \tag{5}$$

Thus $\mathsf{for}\ s\ z\ n$ 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 $\mathsf{for}\ s\ z$ combinator inherits a number of properties stemming from initiality. For instance, the calculation of the following property,

$$s \cdot (\mathsf{for}\ s\ z) = \mathsf{for}\ s\ (s\ z) \tag{6}$$

resorts to “cata-fusion” [5]:

$$
\begin{array}{ll}
 & s \cdot (\mathsf{for}\ s\ z) = \mathsf{for}\ s\ (s\ z) \\
\Leftrightarrow & \quad \{\ (5)\ \text{twice}\ \} \\
 & s \cdot (\!|\, [\underline{z}\,, s] \,|\!) = (\!|\, [\underline{s\ z}\,, s] \,|\!) \\
\Leftarrow & \quad \{\ \text{cata-fusion}\ \} \\
 & s \cdot [\underline{z}\,, s] = [\underline{s\ z}\,, s \cdot s] \\
\Leftrightarrow & \quad \{\ \text{coproducts}\ ;\ \text{constant functions}\ \} \\
 & [\underline{s\ z}\,, s \cdot s] = [\underline{s\ z}\,, s \cdot s] \\
\Box &
\end{array}
$$

2 Calculating for-loops

Although the (iterative) functions that one can write with the **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:

$$
\langle f, g \rangle = (\!|\, \langle h, k \rangle \,|\!) \quad \Leftrightarrow \quad \left\{ \begin{array}{l} f \cdot \mathsf{in} = h \cdot \mathsf{F}\, \langle f, g \rangle \\ g \cdot \mathsf{in} = k \cdot \mathsf{F}\, \langle f, g \rangle \end{array} \right. \tag{7}
$$

In the case of natural numbers, $\mathsf{in} = [\underline{0}, \mathsf{succ}]$ is the Peano algebra of natural numbers, for $\mathsf{F}\ X = 1 + X$.

The law extends to more that 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:

$$
\left\{ \begin{array}{l} \mathit{fac}\ 0 = 0 \\ \mathit{fac}\ (n+1) = (n+1) * \mathit{fac}\ n \end{array} \right.
$$

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

$$
\begin{array}{l}
\mathit{fac}\ n = m\ \textbf{where} \\
\quad (\_, m) = \mathsf{for}\ \mathit{loop}\ (1,1)\ n \\
\quad \mathit{loop}\ (s,n) = (1+s, s*n)
\end{array}
$$

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:

$$
e^x \quad = \quad \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{8}
$$

[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 $\mathsf{exp}\ x\ n$ computes the n-step approximation of e^x:

$$\left\{\begin{array}{l}\mathsf{exp}\ x\ 0 = 1\\ \mathsf{exp}\ x\ (n+1) = h\ x\ n + \mathsf{exp}\ x\ n\end{array}\right.$$
$$\left\{\begin{array}{l}h\ x\ 0 = x\\ h\ x\ (n+1) = \frac{x}{s2\ n}\ (h\ x\ n)\end{array}\right.$$
$$\left\{\begin{array}{l}s2\ 0 = 2\\ s2\ (n+1) = 1 + s2\ n\end{array}\right.$$

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

$$\begin{array}{l}\mathsf{exp}\ x\ n = e\ \mathbf{where}\\ \quad (e,\_,\_) = \mathsf{for}\ loop\ \mathsf{init}\ n\\ \quad \mathsf{init} = (1,2,x)\\ \quad loop\ (e,s,h) = (h+e, s+1, (x\ /\ s) * h)\end{array}$$

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

```
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

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

which leads to a for-loop involving four variables:

$$\begin{array}{l}cos\ x = prj \cdot \mathsf{for}\ loop\ \mathsf{init}\ \mathbf{where}\\ \quad loop\ (c,f,j,k) = (c+f, (-x \uparrow 2\ /\ j) * f, j+k, k+8)\\ \quad \mathsf{init} = (1, -x \uparrow 2\ /\ 2, 12, 18)\\ \quad prj\ (c,\_,\_,\_) = c\end{array}$$

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, $foldr$ comes with its "left"-version $foldl$, which has the same funcionality and implements $foldr$ more efficiently under some mild conditions. What about for? Danvy [1] shows that a "left"-version of 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 $\mathsf{for}\ s\ z = \llparenthesis [\underline{z}, s] \rrparenthesis$ (5) to the pointwise level we get:

$$\left\{ \begin{array}{l} \mathsf{for}\ s\ z\ 0 = z \\ \mathsf{for}\ s\ z\ (n+1) = s\ (\mathsf{for}\ s\ z\ n) \end{array} \right. \tag{9}$$

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

$$\left\{ \begin{array}{l} \mathsf{forl}\ s\ z\ 0 = z \\ \mathsf{forl}\ s\ z\ (n+1) = \mathsf{forl}\ s\ (s\ z)\ n \end{array} \right. \tag{10}$$

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

$$\begin{array}{l} \mathsf{forl}\ s\ 0\ z = z \\ \mathsf{forl}\ s\ (n+1)\ z = \mathsf{forl}\ s\ n\ (s\ z) \end{array}$$

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

$$\begin{array}{l} \mathsf{forl}\ s\ 0 = id \\ \mathsf{forl}\ s\ (n+1) = \lambda z \rightarrow \mathsf{forl}\ s\ n\ (s\ z) \end{array}$$

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 $\mathsf{F}\ X = K^X$ for some non empty K. Functor F is contravariant: given $A \xrightarrow{f} B$, $K^A \xleftarrow{K^f} K^B$ is defined by $K^f = (\cdot f)$.

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

$$k = \mathsf{flip}\, f \ \Leftrightarrow\ f = \underbrace{(\cdot k) \cdot \epsilon}_{\mathsf{flip}\ k} \qquad \begin{tikzcd} K^B & K^{(K^B)} \arrow[d, "(\cdot k)"'] & B \arrow[l, "\epsilon"'] \arrow[dl, "f"] \\ A \arrow[u, "k = \mathsf{flip}\ f"] & K^A & \end{tikzcd} \tag{11}$$

The induced monad is the *continuation monad*, $\mathsf{M}\ X = K^{(K^X)}$. The counit ϵ encodes the continuation style: given $b \in B$ and a continuation f, then $\epsilon\ b\ f = f\ b$. In the diagram, $f\ b = ((\cdot k) \cdot \epsilon)\ b = (\epsilon\ b) \cdot k$ and thus $f\ b\ a = \epsilon\ b\ (k\ a) = k\ a\ b$. That is: $f = \mathsf{flip}\ k$. So isomorphism $\mathsf{flip}\ \cdot$ is its own inverse.

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

$$\mathsf{flip}\ \epsilon = id \tag{12}$$

cancellation,

$$(\cdot \mathsf{flip}\ f) \cdot \epsilon = f \tag{13}$$

fusion,

$$(\mathsf{flip}\ h) \cdot g = \mathsf{flip}\ (K^g \cdot h) \tag{14}$$

absorption,

$$K^g \cdot \mathsf{flip}\ h = \mathsf{flip}\ (h \cdot g) \tag{15}$$

and functor definition:

$$K^h = \mathsf{flip}\ (\epsilon \cdot h) = (\cdot h) \tag{16}$$

Also of interest is the relation between the constant function combinator $\underline{k}$ and the identity:

$$K \xrightarrow{\underline{\cdot}} K^1 \ = \mathsf{flip}\ (\ 1 \xrightarrow{id} K^K\ ) \tag{17}$$

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

$$\begin{tikzcd} \mathsf{L}\ A \to B \arrow[rr, bend left, "\lceil - \rceil"] \arrow[rr, phantom, "\cong"] & & A \to \mathsf{R}\ B \arrow[ll, bend left, "\lfloor - \rfloor"] \end{tikzcd} \tag{18}$$

Let $\mathsf{T} \xleftarrow{\mathsf{in}} \mathsf{F}\ \mathsf{T}$ be an inductive type and $\phi : \mathsf{L}\ \mathsf{F} \to \mathsf{G}\ \mathsf{L}$ be a natural transformation, that is, *free theorem*

$$\phi \cdot (\mathsf{L}\ \mathsf{F}\ k) = (\mathsf{G}\ \mathsf{L}\ k) \cdot \phi \tag{19}$$

holds, for some functor G. Then:

$$f \cdot (\mathsf{L}\ \mathsf{in}) = h \cdot \mathsf{G}\, f \cdot \phi \quad \Leftrightarrow \quad \lceil f \rceil = (\!|\, \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil \,|\!) \tag{20}$$

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

$$\begin{tikzcd}
\mathsf{L}\ \mu_{\mathsf{F}} \arrow[d, "f"'] & & \mathsf{G}\ \mathsf{L}\ \mu_{\mathsf{F}} \arrow[d, "\mathsf{G}\ f"'] & \mathsf{L}\ \mathsf{F}\ \mu_{\mathsf{F}} \arrow[l, "\phi"] \arrow[lll, bend right, "\mathsf{L}\ \mathsf{in}"'] \\
A & & \mathsf{G}\ A \arrow[ll, "h"'] &
\end{tikzcd}$$

$$f = (\!| h |\!) \cdot [\!(\phi \cdot \mathsf{L}\ \mathsf{out})\!]$$

which is converted into its adjoint F-catamorphism:

$$\begin{tikzcd}
\mu_{\mathsf{F}} \arrow[d, "\lceil f \rceil"'] & \mathsf{F}\ \mu_{\mathsf{F}} \arrow[l, "\mathsf{in}"'] \arrow[d, "\mathsf{F}\ \lceil f \rceil"'] \\
\mathsf{R}\ A & \mathsf{F}\ \mathsf{R}\ A \arrow[l, "\lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil"']
\end{tikzcd}$$

$$\begin{tikzcd}
A & \mathsf{G}\ A \arrow[l, "h"'] & \mathsf{G}\ \mathsf{L}\ \mathsf{R}\ A \arrow[l, "\mathsf{G}\ \epsilon"'] & \mathsf{L}\ \mathsf{F}\ \mathsf{R}\ A \arrow[l, "\phi"']
\end{tikzcd}$$

$$\lceil f \rceil = (\!| \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil |\!)$$

Peano adjoint catamorphisms Let us show that $\mathsf{forl}\ s$ is one such hylomorphism. We know:

$\mathsf{F}\ X = 1 + X$
$\mathsf{L}\ X = K^X$
$\lceil f \rceil = \mathsf{flip}\ f$

and so the diagram of f will be (arrows reversed due to contravariance of L):

$$\begin{tikzcd}
K^{\mathbb{N}_0} \arrow[rr, bend left, "(\cdot \mathsf{in})"] & \mathsf{G}\ K^{\mathbb{N}_0} \arrow[r, "\phi"] & K^{1+\mathbb{N}_0} \\
A \arrow[u, "f"] \arrow[r, "h"'] & \mathsf{G}\ A \arrow[u, "\mathsf{G}\ f"] &
\end{tikzcd}$$

We still need to find the recursion pattern $\mathsf{G}\ X$ that is adjoint to the Peano one, $\mathsf{F}\ X = 1 + X$. Since K^{1+X} is isomorphic to $K \times K^X$, it is easy to opt for

$\mathsf{G}\ X = K \times X$

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

$$\phi\ (k, f) = [\underline{k}, f] \qquad (21)$$

The flip of ϕ will be needed below. Let us calculate it by solving

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

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

$$\mathsf{flip}\ \phi = [\underline{\pi_1}\,, \mathsf{flip}\ \pi_2] \tag{22}$$

Thus we can picture, in the diagrams below, the generic theorem (20) tuned to the contravariant exponential adjunction and Peano recursion: hylomorphism f

$$\begin{tikzcd}
K^{\mathbb{N}_0} \arrow[rr, bend left, "(\cdot\mathsf{in})"] & K \times K^{\mathbb{N}_0} \arrow[r, "\phi"] & K^{1+\mathbb{N}_0} \\
A \arrow[u, "f"] \arrow[r, "h"'] & K \times A \arrow[u, "id \times f"'] &
\end{tikzcd}$$

converts into its adjoint F-catamorphism:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "\mathsf{flip}\ f"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + \mathsf{flip}\ f"] \\
K^A & 1 + K^A \arrow[l, "\mathsf{flip}\ (\phi \cdot (id \times \epsilon) \cdot h)"']
\end{tikzcd}$$

$$\begin{tikzcd}
K \arrow[r, "h"] & K \times A \arrow[r, "id \times \epsilon"] & K \times K^{K^A} \arrow[r, "\phi"] & K^{1+K^A}
\end{tikzcd}$$

That is,

$$\mathsf{flip}\ f = \llparenthesis\, \mathsf{flip}\ (\phi \cdot (id \times \epsilon) \cdot h)\, \rrparenthesis \tag{23}$$

equivalent to:

$$f = \mathsf{flip}\ \llparenthesis\, \mathsf{flip}\ (\phi \cdot (id \times \epsilon) \cdot h)\, \rrparenthesis$$

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

$$(\cdot\mathsf{in}) \cdot f = \phi \cdot (id \times f) \cdot h$$

$$\Leftrightarrow \qquad \{\ h = \langle id, s \rangle;\ \text{products}\ \}$$

$$(\cdot\mathsf{in}) \cdot f = \phi \cdot \langle id, f \cdot s \rangle$$

$$\Leftrightarrow \qquad \{\ \text{apply both sides to } z\ \}$$

$$f\ z \cdot \mathsf{in} = \phi\ (z, f\ (s\ z))$$

$$\Leftrightarrow \qquad \{\ \phi\ (k, f) = [\underline{k}\,, f]\ \}$$

$$f\ z \cdot \mathsf{in} = [\underline{z}\,, f\ (s\ z)]$$

$$\Leftrightarrow \qquad \{\ \text{go pointwise}\ \}$$

$$\left\{\begin{array}{l} f\ z\ 0 = z \\ f\ z\ (n+1) = f\ (s\ z)\ n \end{array}\right.$$

$\Leftrightarrow$ { introduce $\mathsf{forl}\ s\ = f$ to make f parametric on s }

$$\left\{\begin{array}{l} \mathsf{forl}\ s\ z\ 0 = z \\ \mathsf{forl}\ s\ z\ (n+1) = \mathsf{forl}\ s\ (s\ z)\ n \end{array}\right.$$

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

$$\mathsf{flip}\ (\mathsf{forl}\ s\ ) = (\!|\ \mathsf{flip}\ (\phi \cdot (id \times \epsilon) \cdot \langle id, s\rangle)\ |\!)$$

We start by simplifying $\mathsf{flip}\ (\phi \cdot (id \times \epsilon) \cdot \langle id, s\rangle)$:

$$\mathsf{flip}\ (\phi \cdot (id \times \epsilon) \cdot \langle id, s\rangle)$$

$=$ { products; absorption (15) }

$$K^{\langle id, \epsilon \cdot s\rangle} \cdot \mathsf{flip}\ \phi$$

$=$ { (22) }

$$K^{\langle id, \epsilon \cdot s\rangle} \cdot [\underline{\pi_1}, \mathsf{flip}\ \pi_2]$$

$=$ { coproducts; constant functions; (22) }

$$[\underline{K^{\langle id, \epsilon \cdot s\rangle}\ \pi_1}, \mathsf{flip}\ (\pi_2 \cdot \langle id, \epsilon \cdot s\rangle)]$$

$=$ { (16); products }

$$[\underline{id}, \mathsf{flip}\ (\epsilon \cdot s)]$$

$=$ { exponential functor (16) }

$$[\underline{id}, (\cdot s)]$$

Thus:

$$\mathsf{flip}\ (\mathsf{forl}\ s\ ) = (\!|\ [\underline{id}, (\cdot s)]\ |\!) = \mathsf{for}\ (\cdot s)\ id \qquad (24)$$

Forl equal to for We finally prove that the equality

$$\mathsf{for}\ = \mathsf{forl}$$

holds:

$$\mathsf{for}\ s\ = \mathsf{forl}\ s$$

$\Leftrightarrow$ { isomorphism $\mathsf{flip}\ \cdot$; (24) }

$$\mathsf{flip}\ (\mathsf{for}\ s\ ) = (\!|\ [\underline{id}, (\cdot s)]\ |\!)$$

$\Leftrightarrow$ { cata-universal; (15) }

$$\left\{\begin{array}{l} \mathsf{flip}\ (\mathsf{for}\ s\ ) \cdot \underline{0} = \underline{id} \\ \mathsf{flip}\ (\mathsf{for}\ s\ ) \cdot \mathsf{succ} = \mathsf{flip}\ (\mathsf{for}\ s\ \cdot s) \end{array}\right.$$

$\Leftrightarrow$ { (17); drop isomorphism $\mathsf{flip}\ \cdot$ }

$$\left\{\begin{array}{l} (\cdot\underline{0}) \cdot \mathsf{for}\ s\ = \underline{\cdot} \\ (\cdot\mathsf{succ}) \cdot \mathsf{for}\ s\ = \mathsf{for}\ s\ \cdot s \end{array}\right.$$

$\Leftrightarrow$ $\{$ apply all terms to z and simplify $\}$

$$\left\{\begin{array}{l} \mathsf{for}\ s\ z \cdot \underline{0} = \underline{z} \\ \mathsf{for}\ s\ z \cdot \mathsf{succ} = \mathsf{for}\ s\ (s\ z) \end{array}\right.$$

$\Leftrightarrow$ $\{$ (6) $\}$

$$\left\{\begin{array}{l} \mathsf{for}\ s\ z \cdot \underline{0} = \underline{z} \\ \mathsf{for}\ s\ z \cdot \mathsf{succ} = s \cdot \mathsf{for}\ s\ z \end{array}\right.$$

$\Leftrightarrow$ $\{$ cata-universal $\}$

$\mathsf{for}\ s\ z = (\!|\, [\underline{z}\,, s] \,|\!)$

$\Leftrightarrow$ $\{$ (5) $\}$

$true$

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

[1] Olivier Danvy. Folding left and right over peano numbers. *J. Funct. Program.*, 29:e6, 2019.

[2] M. Erwig and S. Kollmannsberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

[3] R. Hinze. Adjoint folds and unfolds — an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.

[4] A. Neri, R.S. Barbosa, and J.N. Oliveira. Compiling quantamorphisms for the IBM Q-Experience. 2020. Submitted to IEEE Trans. Soft. Eng.

[5] J.N. Oliveira. Program Design by Calculation, 2019. Draft of textbook in preparation, current version: Feb. 2021. Informatics Department, University of Minho (PDF[2]).

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