On Barron and Strachey's Cartesian Product Function

Olivier Danvy
Michael Spivey

BRICS Report Series RS-07-14

ISSN 0909-0878 July 2007

On Barron and Strachey's Cartesian Product Function

Possibly the world's first functional pearl[*]

Olivier Danvy
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark[†]

Michael Spivey
Computing Laboratory
Oxford University
Wolfson Building, Parks Road
Oxford OX1 3QD, England[‡]

July 20, 2007

Abstract

Over forty years ago, David Barron and Christopher Strachey published a startlingly elegant program for the Cartesian product of a list of lists, expressing it with a three nested occurrences of the function we now call *foldr*. This program is remarkable for its time because of its masterful display of higher-order functions and lexical scope, and we put it forward as possibly the first ever functional pearl. We first characterize it as the result of a sequence of program transformations, and then apply similar transformations to a program for the classical power set example. We also show that using a higher-order representation of lists allows a definition of the Cartesian product function where *foldr* is nested only twice.

[*]To appear in the proceedings of ICFP'07.
[†]E-mail: `danvy@brics.dk`
[‡]E-mail: `mike@comlab.ox.ac.uk`

Contents

1 Introduction **1**

2 Cartesian product explained **3**

3 Cartesian product reconstructed **4**

4 Application to the power set function **7**

5 Cartesian product revisited **8**

6 In conclusion **12**

1 Introduction

In 1966, David Barron and Christopher Strachey contributed a chapter on 'Programming' to a book entitled *Advances in Programming and Non-numerical Computation* edited by Leslie Fox, then Professor of Numerical Analysis at Oxford (Barron and Strachey 1966). The volume was assembled from lecture notes used at a summer school held in Oxford in 1963, and Barron and Strachey's chapter was put together by David Barron with the help of tape recordings of Strachey's lectures (Barron 1975). Although ostensibly an introduction to (then) modern ideas in programming, the chapter could more accurately be described as a shop window for the features of the authors' new programming language CPL.

CPL was far ahead of its time, in the sense that it was too ambitious ever to be fully implemented with contemporary machines and software (Hartley 2000). Partly through its simpler variant BCPL (Richards 2000), introduced as a language in which the CPL compiler could be written, CPL did have a wide influence, however. BCPL in turn gave rise to the languages B and C in which the UNIX system was written, and so its indirect influence remains widespread even today.

Barron and Strachey's chapter contains several examples of CPL programs in different styles, and among them is a purely functional program for computing the list of all factors of a given number. In the best functional style, this program is a composition of simpler parts:

1. Use repeated division to find a list of prime factors of the given number in ascending order.

2. Group equal factors and multiply them together to get lists of the prime powers that divide the given number.

3. Use a Cartesian product function to choose one of the powers of each prime in each possible way.

4. Multiply together the prime powers to give all the factors of the number.

What interests us here is the Cartesian product function, which Barron and Strachey introduce with the example that $\mathit{Product}$ applied to the list $[[a, b], [p, q, r], [x, y]]$ yields

$$\begin{aligned}&[[a,p,x],[a,p,y],[a,q,x],[a,q,y],[a,r,x],[a,r,y],\\&\quad[b,p,x],[b,p,y],[b,q,x],[b,q,y],[b,r,x],[b,r,y]],\end{aligned}$$

with each list in the result containing one element from each of the lists in the input, and in the same order. Later, they provide the following startling definition of this function:

```
let $\mathit{Product}[L] = \mathit{Lit}[f, \mathit{List}1[\mathrm{NIL}], L]$
  where $f[k, z] = \mathit{Lit}[g, \mathrm{NIL}, k]$
    where $g[x, y] = \mathit{Lit}[h, y, z]$
      where $h[p, q] = \mathit{Cons}[\mathit{Cons}[x, p], q]$.
```

Here, *Lit* is the higher-order function that present-day functional programmers call *foldr*:

```
$\mathit{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$
$\mathit{foldr}\ f\ a\ \mathit{xs} = \mathit{visit}\ \mathit{xs}$
  where
    $\mathit{visit}\ [\,] = a$
    $\mathit{visit}\ (x : \mathit{xs}) = f\ x\ (\mathit{visit}\ \mathit{xs})$
```

Using some more notation from Haskell together with the '*xs*' convention for list variables, the program can be rewritten as follows:

```
$\mathit{product} :: [[\alpha]] \to [[\alpha]]$                                        $(P)$
$\mathit{product}\ \mathit{xss} = \mathit{foldr}\ f\ [[\,]]\ \mathit{xss}$
  where $f\ \mathit{xs}\ \mathit{yss} = \mathit{foldr}\ g\ [\,]\ \mathit{xs}$
    where $g\ x\ \mathit{zss} = \mathit{foldr}\ h\ \mathit{zss}\ \mathit{yss}$
      where $h\ \mathit{ys}\ \mathit{qss} = (x : \mathit{ys}) : \mathit{qss}$.
```

(We have compressed the layout a little to save space.)

This definition of *Product* is astounding in several ways, and (although it is a subjective point) we put it forward as possibly the first ever functional pearl, in the sense of a presentation of a purely functional computer program that is remarkable for its succinct elegance. It makes accomplished use of higher-order functions (the three nested occurrences of *foldr*), and of lexical scoping (the occurrence of x in the last line is bound by the enclosing definition "**where** $g\ x\ \mathit{zss} = \ldots$", and the occurrence of *yss* in the third line is similarly bound by "**where** $f\ \mathit{xs}\ \mathit{yss} = \ldots$").

Though higher-order functions such as *map* were known in Lisp, which had a clear influence on Strachey's work of the period, Lisp at the time lacked any equivalent for the function *foldr*, had dynamic scope by default, and did not readily support local function definitions. The archive of Strachey's working papers at the Bodleian Library in Oxford (Strachey 1961) contains many versions of the Cartesian product function, beginning in late 1961; it was evidently one of Strachey's favourite examples. A number of listings show him testing various versions by translating them into Lisp.

However, in a handwritten note dated 2 November 1961, Strachey goes beyond the Lisp style of the time and investigates the properties of two "recursive operators" R_1 and R_2 that correspond to *foldr* and *foldl*. Strachey first notes that R_1 applied to *cons* has the effect of *append* and R_2 applied to *cons* has the effect of *reverse*. He then shows that the Cartesian product function can also be expressed in terms of R_1 using nested lambda-expressions where free variables of the inner expression are bound by the outer lambda, giving in essence the program shown above. Though this program could have been translated into Lisp using the FUNARG mechanism, there is no evidence that Strachey did so.

In this article, we reconstruct a sequence of steps that might have led to Barron and Strachey's program, relate these to a more 'modern' derivation that starts with a list comprehension, and treat another example – power set – in a similar way, before finally showing how a program of still higher order can compute *product* using only two applications of *foldr*.

As for the authors of the present article, Danvy first came across this arrestingly beautiful definition of the Cartesian product in Patrick Greussay's VLISP 16 manual (1978, page 3–3), and Spivey in Fox's book (Barron and Strachey 1966). Naturally, we were not the first to marvel at the programming achievements described in Barron and Strachey's chapter and epitomized by the definition of *product* shown above. Michael Gordon (1973; 1979; 2000) showed that any function that is defined using *Lit* but no other recursion does an amount of work that is bounded as a function of the length of the longest list in the input. This allowed him to show that *Lit* cannot be used to define many familiar functions on trees (represented as nested lists), where the input may be deeply nested but contains only lists of bounded length.

2 Cartesian product explained

Barron and Strachey do not quite pull out of a hat the definition of *Product* from Section 1. Noting that the problem of defining *Product* is "quite difficult", they begin with the following definition, which we have re-expressed using pattern matching for clarity.

$$\begin{aligned}&\mathit{product}\,[\,] = [[\,]] \qquad\qquad (P_0)\\ &\mathit{product}\,([\,]:\_) = [\,]\end{aligned}$$

```
$product\ ((x : xs) : xss) =$
  $map\ f\ (product\ xss) \mathbin{+\!\!+} product\ (xs : xss)$
    **where** $f\ ys = x : ys.$
```

The authors show a trace of the execution of the program and note, "Although this program is ingenious, this is a very inefficient process," before proposing the following "more efficient" version.[1]

```
$product\ [\,] = [[\,]]$                                         $(P_1)$
$product\ (xs : xss) = f_1\ xs\ (product\ xss)$
  **where**
    $f_1\ [\,]\ yss = [\,]$
    $f_1\ (x : xs)\ yss = f_2\ x\ (f_1\ xs\ yss)\ yss$
      **where**
        $f_2\ x\ zss\ [\,] = zss$
        $f_2\ x\ zss\ (ys : yss) = (x : ys) : f_2\ x\ zss\ yss.$
```

Then they write, "This process can be expressed more elegantly" in the form shown in Section 1 of the present article, leaving the reader with no more than an elliptic explanation of this *tour de force*.

3 Cartesian product reconstructed

In this section, we present a sequence of transformation steps that leads from the "ingenious but inefficient" version P_0 to the "more efficient" stepping stone P_1 and on to the definition in terms of $foldr$.

Introducing an auxiliary function

The function $product$ in program P_0 uses recursion on both the list of lists and on the list that is its first element. Let us introduce an auxiliary function h to separate the two recursions, specifying it by

$$h\ xs\ xss = product\ (xs : xss).$$

Disentangling the program in this way leads to the following version of $product$:

[1]In transcribing the definition of f_1, we have replaced the expression $f_2\ x\ zss\ (f_1\ xs\ zss)$, an equivalent of which appeared in the original paper, with the corrected expression $f_2\ x\ (f_1\ xs\ zss)\ zss$.

$$
\begin{array}{l}
\mathit{product}\,[\,] = [[\,]] \\
\mathit{product}\,(\mathit{xs} : \mathit{xss}) = h\,\mathit{xs}\,\mathit{xss} \\
\quad \textbf{where} \\
\qquad h\,[\,]\,\mathit{xss} = [\,] \\
\qquad h\,(x : \mathit{xs})\,\mathit{xss} = \mathit{map}\,f\,(\mathit{product}\,\mathit{xss}) +\!\!+\, h\,\mathit{xs}\,\mathit{xss} \\
\qquad\quad \textbf{where}\ f\,\mathit{ys} = x : \mathit{ys}.
\end{array}
\tag{P_0'}
$$

Computing the main recursive call just once

In program P_0', the recursive call $\mathit{product}\ \mathit{xss}$ is computed repeatedly, since xss is an unchanging argument of h. It is better to compute this call just once, and we arrange for this by replacing h with a new function h', specified by

$$h'\,\mathit{xs}\,(\mathit{product}\,\mathit{xss}) = h\,\mathit{xs}\,\mathit{xss}.$$

This replacement leads to the following rearrangement of the program:

$$
\begin{array}{l}
\mathit{product}\,[\,] = [[\,]] \\
\mathit{product}\,(\mathit{xs} : \mathit{xss}) = h'\,\mathit{xs}\,(\mathit{product}\,\mathit{xss}) \\
\quad \textbf{where} \\
\qquad h'\,[\,]\,\mathit{yss} = [\,] \\
\qquad h'\,(x : \mathit{xs})\,\mathit{yss} = \mathit{map}\,f\,\mathit{yss} +\!\!+\, h'\,\mathit{xs}\,\mathit{yss} \\
\qquad\quad \textbf{where}\ f\,\mathit{ys} = x : \mathit{ys}.
\end{array}
\tag{P_0''}
$$

Since the value of $\mathit{product}\ (\mathit{xs} : \mathit{xss})$ depends on the value of $\mathit{product}\ \mathit{xss}$, we can view the result of $\mathit{product}$ in this program as a synthesized attribute (Johnsson 1987).

This form of $\mathit{product}$ is easily reached also by beginning with a definition that uses generators in the form of a list comprehension:

$$
\begin{array}{l}
\mathit{product}\,[\,] = [[\,]] \\
\mathit{product}\,(\mathit{xs} : \mathit{xss}) = [\,x : \mathit{ys} \mid x \leftarrow \mathit{xs}, \mathit{ys} \leftarrow \mathit{product}\,\mathit{xss}\,].
\end{array}
$$

If we define $h'\,\mathit{xs}\,\mathit{yss} = [\,x : \mathit{ys} \mid x \leftarrow \mathit{xs}, \mathit{ys} \leftarrow \mathit{yss}\,]$, then we immediately get the equation

$$\mathit{product}\,(\mathit{xs} : \mathit{xss}) = h'\,\mathit{xs}\,(\mathit{product}\,\mathit{xss}).$$

Applying the laws

$$[\,E \mid p \leftarrow [\,], q \leftarrow \mathit{ys}\,] = [\,]$$

and

$$\begin{aligned}&[\,E \mid p \leftarrow x : xs, y \leftarrow ys\,]\\ &\quad = [\,E[x/p] \mid y \leftarrow ys\,] \mathbin{+\!\!+} [\,E \mid p \leftarrow xs, q \leftarrow ys\,]\end{aligned}$$

then gives the recursive definition of h'.

Eliminating *append*

The definition of h' includes the equation,

$$h'\,(x : xs)\;yss = map\;f\;yss \mathbin{+\!\!+} h'\;xs\;yss.$$

We can eliminate the use of $\mathbin{+\!\!+}$ by introducing a function p, specified by

$$p\,x\;zss\;yss = map\;f\;yss \mathbin{+\!\!+} zss\;\textbf{where}\;f\;ys = x : ys$$

Now we can replace the expression $map\;f\;yss \mathbin{+\!\!+} h'\;xs\;yss$ with $p\,x\,(h'\;xs\;yss)\;yss$. We can also derive a recursive definition of p:

$$p\,x\;zss\;[\,] = zss,$$

$$\begin{aligned}&p\,x\;zss\;(ys : yss)\\ &\quad = f\;ys : map\;f\;yss \mathbin{+\!\!+} zss\\ &\quad = (x : ys) : p\,x\;zss\;yss.\end{aligned}$$

Putting these definitions together, and renaming h' as f_1 and p as f_2, gives the program P_1 that appears in Section 2.[2]

Introducing *foldr*

The next step is to observe that in program P_1,

```
$product\;[\,] = [[\,]]$                                    $(P_1)$
$product\;(xs : xss) = f_1\;xs\;(product\;xss)$
  **where**
    $f_1\;[\,]\;yss = [\,]$
    $f_1\;(x : xs)\;yss = f_2\;x\;(f_1\;xs\;yss)\;yss$
      **where**
        $f_2\;x\;zss\;[\,] = zss$
```

[2]The idea of introducing an extra parameter in order to eliminate $\mathbin{+\!\!+}$ is a recurring theme in Strachey's working papers, often expressed in terms of a 'deforested' function $mapa\;f\;xs\;ys$ that computes $map\;f\;xs \mathbin{+\!\!+} ys$.

$$f_2\, x\, zss\, (ys : yss) = (x : ys) : f_2\, x\, zss\, yss,$$

each of *product*, f_1 and f_2 can be rewritten using *foldr*. First,

$$product\ xss = foldr\ f_1\ [[\,]]\ xss,$$

and second,

$$f_1\ xs\ yss = foldr\ (g\ yss)\ [\,]\ xs,$$

where $g\ yss\ x\ zss = f_2\ x\ zss\ yss$. Third, we see that

$$\begin{array}{l} g\ yss\ x\ zss = foldr\ (h\ x)\ zss\ yss \\ \quad \textbf{where}\ h\ x\ ys\ qss = (x : ys) : qss. \end{array}$$

Exploiting nested scopes

The final step is to note that the argument yss of f_1 is passed on unchanged as an argument of g, and the argument x of g is passed on as an argument of h. There is no need to make these arguments explicit, and they can be 'lambda-dropped' (Danvy and Schultz 2000) and left as free variables of g and h respectively. Renaming f_1 as f then gives the form of the program P that was shown in Section 1.

4 Application to the power set function

A similar sequence of transformation steps can be applied to other functions. For example, let us consider the power set function defined by

$$\begin{array}{l} powerset :: [\alpha] \to [[\alpha]] \\ powerset\ [\,] = [[\,]] \\ powerset\ (x : xs) = map\ (x{:})\ yss \mathbin{+\!\!+} yss \\ \quad \textbf{where}\ yss = powerset\ xs. \end{array} \qquad (Q_0)$$

Applying it to the list $[a, b, c]$ yields

$$[[a, b, c], [a, b], [a, c], [a], [b, c], [b], [c], [\,]].$$

As with the version P_0' of the Cartesian product function, this definition contains a use of $\mathbin{+\!\!+}$ that can be eliminated by introducing a new function, in this case specified by

$$f\ x\ yss\ zss = map\ (x{:})\ yss +\!\!+\ zss.$$

The second equation in Q_0 then becomes

$$\begin{aligned}&powerset\ (x : xs) = f\ x\ yss\ yss\\&\quad \textbf{where}\ yss = powerset\ xs,\end{aligned}$$

and we can derive the recursive definition,

$$\begin{aligned}&f\ x\ [\,]\ zss = zss\\&f\ x\ (ys : yss)\ zss = (x : ys) : f\ x\ yss\ zss.\end{aligned}$$

The program can now be expressed in terms of $foldr$: giving

$$powerset\ xs = foldr\ g\ [[\,]]\ xs$$

where $g\ x\ yss = f\ x\ yss\ yss$, and

$$f\ x\ yss\ zss = foldr\ (h\ x)\ zss\ yss$$

where $h\ x\ ys\ qss = (x : ys) : qss$.

When we put the results of these calculations together, x can be lambda-dropped, and we obtain a program in the style of Barron and Strachey:

$$\begin{aligned}&powerset\ xs = foldr\ g\ [[\,]]\ xs\\&\quad \textbf{where}\ g\ x\ yss = foldr\ h\ yss\ yss\\&\qquad \textbf{where}\ h\ ys\ qss = (x : ys) : qss.\end{aligned} \qquad (Q_1)$$

Of course, we could also have used $mapa$ here to start with (see Footnote 2). In any case, this program is essentially the same as one attributed by Gordon (1973; 1979) to his colleague Dave du Feu.

5 Cartesian product revisited

Consider for a moment a function that computes the Cartesian product of just four lists:

$$\begin{aligned}&\mathit{product\_of\_four} :: [a] \to [a] \to [a] \to [a] \to [[a]]\\&\mathit{product\_of\_four}\ xs\ ys\ zs\ ws =\\&\quad concat\ (map\ f\ xs)\\&\qquad \textbf{where}\ f\ x = concat\ (map\ g\ ys)\\&\qquad\quad \textbf{where}\ g\ y = concat\ (map\ h\ zs)\end{aligned}$$

$$\textbf{where}\ h\,z = concat\,(map\,k\,ws)$$
$$\quad\textbf{where}\ k\,w = [[x,y,z,w]].$$

This program is exactly the result of translating the list comprehension

$$[\,[x,y,z,w] \mid x \leftarrow xs, y \leftarrow ys, z \leftarrow zs, w \leftarrow ws\,].$$

Let us think about the purpose of the local function g that comes in the middle of the nest of five functions. For fixed values of x and y, it computes the list of all lists $[x,y,z,w]$ where z and w are respectively drawn from the lists zs and ws: in other words,

$$map\,([x,y]{+\!\!+})\,(\mathit{product\_of\_two}\,zs\,ws),$$

where $([x,y]{+\!\!+})$ is the function that maps any list ps to the list $[x,y] \mathbin{+\!\!+} ps$.

This observation suggests that it might be fruitful to consider a recursive definition of a function that, given xss and us, computes

$$map\,(us{+\!\!+})\,(product\,xss).$$

Even better, we can exploit an idea of Hughes (1986), and represent the list us by the function $h = (us{+\!\!+})$. Thus, we make the specification,

$$prod :: [[\alpha]] \to ([\alpha] \to [\alpha]) \to [[\alpha]]$$
$$prod\,xss\,h = map\,h\,(product\,xss),$$

and calculate a recursive definition of $prod$ as follows:

$$\begin{aligned}
&prod\,[\,]\,h\\
&\quad = map\,h\,(product\,[\,])\\
&\quad = [h\,[\,]],\\
&prod\,(xs : xss)\,h\\
&\quad = map\,h\,(product\,(xs : xss))\\
&\quad = f\,xs\,(prod\,xss)\,h,
\end{aligned}$$

where

$$f :: [\alpha] \to (([\alpha] \to [\alpha]) \to [[\alpha]]) \to ([\alpha] \to [\alpha]) \to [[\alpha]]$$

is a function such that if $c = prod\,xss$ then

$$f\,xs\,c\,h = map\,h\,(product\,(xs : xss)).$$

We can immediately calculate a clause that defines f when $xs = [\,]$:

$$\begin{aligned}
&f\,[\,]\,c\,h\\
&\quad = map\,h\,(product\,([\,] : xss))\\
&\quad = map\,h\,[\,]\\
&\quad = [\,].
\end{aligned}$$

To derive the second clause in a definition of f, we must use the equation

$$\begin{aligned}
&product\,((x : xs) : xss) =\\
&\quad map\,(x{:})\,(product\,xs) +\!\!+\; product\,(xs : xss),
\end{aligned}$$

together with the laws $map\,h\,(ps +\!\!+\; qs) = map\,h\,ps +\!\!+\; map\,h\,qs$ and $map\,h\,(map\,g)\,zss = map\,(h \cdot g)\,zss$:

$$\begin{aligned}
&f\,(x : xs)\,c\,h\\
&\quad = map\,h\,(product\,((x : xs) : xss))\\
&\quad = map\,h\,(map\,(x{:})\,(product\,xss)) +\!\!+\; map\,h\,(product\,(xs : xss))\\
&\quad = prod\,xss\,(h \cdot (x{:})) +\!\!+\; f\,xs\,c\,h\\
&\quad = c\,(h \cdot (x{:})) +\!\!+\; f\,xs\,c\,h.
\end{aligned}$$

Here, $h \cdot (x{:})$ denotes the composition of h with the function $(x{:})$ that adds the element x at the front of a list. Thus $h \cdot (x{:})$ is the function h' such that $h'\,xs = h\,(x : xs)$. In contrast to the synthesized attributes that were present in the successive versions of *product* in Section 3, the function h plays the role of an inherited attribute here.

In summary, we have derived the program,

```
product xss = prod xss id
  where
    prod [] h = [h []]
    prod (xs : xss) h = f xs (prod xss) h
      where
        f [] c h = []
        f (x : xs) c h = c (h · (x:)) ++ f xs c h,
```

which gives exactly the same result as Barron and Strachey's program (P) in Section 1. If we were to use a first-order representation of lists here, we would naturally define a version of the *product* function where the individual sublists of the result appeared in reverse:

```
product xss = prod xss []
```

```
  where
    $prod\,[\,]\,h' = [reverse\,h']$
    $prod\,(xs : xss)\,h' = f\,xs\,(prod\,xss)\,h'$
      where
        $f\,[\,]\,c\,h' = [\,]$
        $f\,(x : xs)\,c\,h' = c\,(x : h') +\!\!+ f\,xs\,c\,h',$
```

And indeed, and Danvy and Nielsen point out (2001), defunctionalising Hughes' representation gives this first-order representation, including an application of *reverse* in the right place.

Once again, we would like to play the trick of eliminating $+\!\!+$ from this program, but this is made a little more difficult by the presence of the functional argument $c :: ([\alpha] \to [\alpha]) \to [[\alpha]]$. To deal with this, suppose that

$$c' :: ([\alpha] \to [\alpha]) \to [[\alpha]] \to [[\alpha]]$$

is related to c by the equation

$$c'\,h\,zss = c\,h +\!\!+ zss$$

which holds for all zss; we specify a new pair of functions $prod'$ and f' by the equations,

$$prod' :: [[\alpha]] \to ([\alpha] \to [\alpha]) \to [[\alpha]] \to [[\alpha]]$$
$$prod'\,xss\,h\,zss = prod\,xss\,h +\!\!+ zss$$

$$f' :: [\alpha] \to (([\alpha] \to [\alpha]) \to [[\alpha]]) \to$$
$$([\alpha] \to [\alpha]) \to [[\alpha]] \to [[\alpha]]$$
$$f'\,xs\,c'\,h\,zss = f\,xs\,c\,h +\!\!+ zss$$

Now we can calculate,

$$prod'\,[\,]\,h\,zss = h\,[\,] : zss$$

$$\begin{aligned} &prod'\,(xs : xss)\,h\,zss \\ &\quad = f\,xs\,(prod\,xss)\,h +\!\!+ zss \\ &\quad = f'\,xs\,(prod'\,xss)\,h\,zss \end{aligned}$$

$$f'\,[\,]\,c'\,h\,zss = zss$$

$$\begin{aligned} &f'(x : xs)\,c'\,h\,zss \\ &\quad = c\,(h \cdot (x{:})) +\!\!+ f\,xs\,c\,h +\!\!+ zss \\ &\quad = c'\,(h \cdot (x{:}))\,(f'\,xs\,c'\,h\,zss). \end{aligned}$$

In the end, this apparently circular derivation is justified by induction on the list structure of the input. We have thus derived the recursive definition,

$$
\begin{array}{l}
\mathit{prod}'\ [\,]\ h\ \mathit{zss} = h\ [\,] : \mathit{zss} \\
\mathit{prod}'\ (\mathit{xs} : \mathit{xss})\ h\ \mathit{zss} = f'\ \mathit{xs}\ (\mathit{prod}'\ \mathit{xss})\ h\ \mathit{zss} \\
\quad \textbf{where} \\
\qquad f'\ [\,]\ c'\ h\ \mathit{zss} = \mathit{zss} \\
\qquad f'\ (x : \mathit{xs})\ c'\ h\ \mathit{zss} = c'\ (h \cdot (x{:}))\ (f'\ \mathit{xs}\ c'\ h\ \mathit{zss}).
\end{array}
$$

This version is ready to be expressed in terms of foldr. By writing $\mathit{prod}' = \mathit{foldr}\ f'\ u$ for a suitable function u, then putting $\mathit{product}\ \mathit{xss} = \mathit{prod}'\ \mathit{xss}\ \mathit{id}\ [\,]$, we obtain

$$
\begin{array}{l}
\mathit{product}\ \mathit{xss} = \mathit{foldr}\ f'\ u\ \mathit{xss}\ \mathit{id}\ [\,] \\
\quad \textbf{where} \\
\qquad u\ h\ \mathit{zss} = h\ [\,] : \mathit{zss} \\
\qquad f'\ \mathit{xs}\ c'\ h\ \mathit{zss} = \mathit{foldr}\ g\ \mathit{zss}\ \mathit{xs} \\
\qquad\quad \textbf{where}\ g\ x\ \mathit{qss} = c'\ (h \cdot (x{:}))\ \mathit{qss}.
\end{array}
\tag{P_2}
$$

This version of $\mathit{product}$ gives exactly the same result as Barron and Strachey's program (P), but contains only two occurrences of foldr, corresponding to the two levels of list structure in the argument, and uses only inherited attributes. The extra mileage comes from our more extensive use of higher-order functions.

6 In conclusion

Was Barron and Strachey's Cartesian product function really the first ever functional pearl? There are, admittedly, still earlier pearls of insight by others that deserve to be celebrated: notably the work of Church on lambda-definability, of Curry on Combinatory Logic, and of McCarthy on Lisp. Nevertheless, the work we have explored in this article is distinctive in its own right, for Barron and Strachey were writing explicitly about computer programs as objects of study. They exploited the expressive possibilities of a higher-order, purely functional style, both in writing the Cartesian product function itself, and in using it as part of a larger program for finding factors. The lectures at that 1963 summer school must have been bewildering to some members of Barron and Strachey's audience, many of them perhaps brought up on a diet of Autocode, and only dimly aware of the new ideas in Lisp and Algol 60. Yet the lessons that were taught then are still worth learning today, over forty years later.

Acknowledgments: We are grateful to the staff of the Bodleian Library in Oxford for their friendly efficiency, and to the ICFP reviewers for their helpful comments.

References

David Barron. 1975. Christopher Strachey: a personal reminiscence. *Computer Bulletin*, 2(5):8–9.

David W. Barron and Christopher Strachey. 1966. Programming. In Leslie Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 49–82. Pergammon Press.

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy. ACM Press. Extended version available as the research report BRICS RS-01-23.

Olivier Danvy and Ulrik P. Schultz. 2000. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287. A preliminary version was presented at the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1997).

Michael J. C. Gordon. 1973. An investigation of lit. Technical Report Memorandun MIP-R-101, School of Artificial Intelligence, University of Edinburgh.

Michael J. C. Gordon. 1979. On the power of list iteration. *The Computer Journal*, 22(4):376–379.

Mike Gordon. 2000. Christopher Strachey: recollections of his influence. *Higher-Order and Symbolic Computation*, 13(1/2):65–67.

Patrick Greussay. 1978. Le système VLISP 16. Université Paris-8-Vincennes et LITP.

David Hartley. 2000. Cambridge and CPL in the 1960s. *Higher-Order and Symbolic Computation*, 13(1/2):69–70.

John Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144.

Thomas Johnsson. 1987. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 154–173, Portland, Oregon. Springer-Verlag.

Martin Richards. 2000. Christopher strachey and the Cambridge CPL compiler. *Higher-Order and Symbolic Computation*, 13(1/2):85–88.

Christopher Strachey. 1961. Handwritten notes. Archive of working papers and correspondence. Bodleian Library, Oxford, Catalogue no. MS. Eng. misc. b.267.

Recent BRICS Report Series Publications