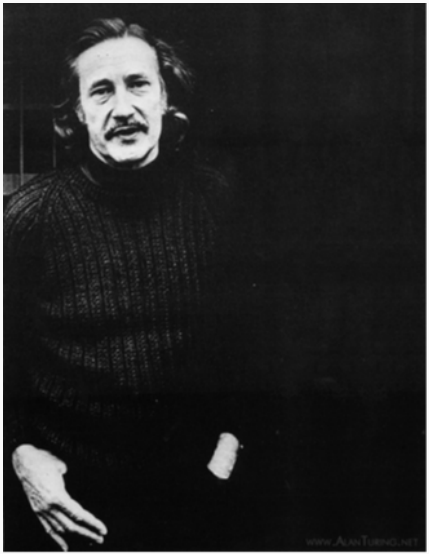
On the Correctness of Barron and Strachey's Cartesian Product Function

joint work with Jason Hemann

Wouter Swierstra

Utrecht University

Christopher Strachey



- First professor of Computer Science in Oxford;
- Pioneering work in denotational semantics, functional programming, and many other areas;
- Wrote the first natural language generator – for love letters!
- Wrote the first program to play music...
- And the first computer game!

‘Programming’ Chapter in *Advances in Programming and Non-numerical Computation* (1966)

Strachey designed ‘Combined Programming Language’ (CPL), an early ancestor of C.

This book chapter, together with David Barron, became a ‘CPL primer’.

Among other things, it contains the following function:

```
let P [L] = Lit[f, List1[NIL], L]
  where f [k, z] = Lit[g, NIL, k]
    where g[x, y] = Lit[h, y, z]
      where h[p, q] = Cons[Cons[x, p], q]
```

What does this do?

In Haskell

```
p :: [[α]] → [[α]]
p xss = foldr f [[]] xss
  where f xs yss = foldr g [] xs
    where g x zss = foldr h zss yss
      where h ys qss = (x : ys) : qss
```

In Haskell

```
p :: [[α]] → [[α]]
p xss = foldr f [[]] xss
  where f xs yss = foldr g [] xs
    where g x zss = foldr h zss yss
      where h ys qss = (x : ys) : qss
```

This compute the *Cartesian product* of a list of lists.

Example

```
Main> product [[1,2],[3,4,5],[6,7]]
  [[1 , 3 , 6 ], [1 , 3 , 7 ],
   [1 , 4 , 6 ], [1 , 4 , 7 ],
   [1 , 5 , 6 ], [1 , 5 , 7 ],
   [2 , 3 , 6 ], [2 , 3 , 7 ],
   [2 , 4 , 6 ], [2 , 4 , 7 ],
   [2 , 5,  6 ], [2 , 5 , 7]]
```

Cartesian product

```
product :: [[α]] → [[α]]
product xss = foldr f [[]] xss
  where f xs yss = foldr g [] xs
    where g x zss = foldr h zss yss
      where h ys qss = (x : ys) : qss
```

This function uses lexical scoping, polymorphism, higher-order functions, …

Cartesian product

```
product :: [[α]] → [[α]]
product xss = foldr f [[]] xss
  where f xs yss = foldr g [] xs
    where g x zss = foldr h zss yss
      where h ys qss = (x : ys) : qss
```

This function uses lexical scoping, polymorphism, higher-order functions, …

Quite innovative fifty years ago!

On Barron and Strachey's Cartesian Product Function (2007)

Olivier Danvy and Michael Spivey wrote a lovely paper on what they call 'possibly the world's first functional pearl'.

They show how to derive this function through a series of program transformations, starting from a simpler (but inefficient) version:

```
product []                = [[]]
product ([] : xss)        = []
product ((x : xs) : xss) = map (x:) (product xss) ++ product (xs : xss)
```

What else could there possibly be to say?

On Barron and Strachey's Cartesian Product Function (2007)

Olivier Danvy and Michael Spivey wrote a lovely paper on what they call 'possibly the world's first functional pearl'.

They show how to derive this function through a series of program transformations, starting from a simpler (but inefficient) version:

```
product []                = [[]]
product ([] : xss)        = []
product ((x : xs) : xss) = map (x:) (product xss) ++ product (xs : xss)
```

What else could there possibly be to say?

Shall we trying proving the tricky version correct?

On program correctness

To do any kind of verification we need three things:

- a program;
- a specification;
- a proof relating the two.

On program correctness

To do any kind of verification we need three things:

- a program;
- a specification;
- a proof relating the two.

What is our specification?

What makes a good spec?

We *could* take a simpler definition of the cartesian product function as the *reference implementation* and use that for our specification:

- we now assume our reference implementation is correct…
- doing so may be overly restrictive: for example, this *fixes* the order of elements in the cartesian product – which may not be what we want!

What makes a good spec?

We *could* take a simpler definition of the cartesian product function as the *reference implementation* and use that for our specification:

- we now assume our reference implementation is correct…
- doing so may be overly restrictive: for example, this *fixes* the order of elements in the cartesian product – which may not be what we want!

Instead it can be useful to think more abstractly over the specification.

Relational specification

We will define a relation between the input and output lists:

```
correct :: [[a]] -> [[a]] -> Bool
```

- soundness: each list in the result, draws its elements from the corresponding input list;
- completeness: any such list, occurs in the result.

Pairwise relations

To define these relations, we will need one auxiliary function (and a few from the Prelude):

```
pairwise :: (a -> b -> Bool) -> ([a] -> [b] -> Bool)
pairwise p []       []       = True
pairwise p (x:xs)   (y:ys)   = p x y && pairwise p xs ys
pairwise p _        _        = False
```

Note: this is *always false* if the lists have different lengths.

Soundness

Each element of an output list most be drawn from the corresponding input list:

```
soundness :: Eq a => [[a]] -> [[a]] -> Bool
soundness xss = all (\ys -> pairwise elem ys xss)
```

From relations to tests…

We define the corresponding QuickCheck test straight away:

```
satisfies :: (a -> b) -> (a -> b -> c) -> (a -> c)
satisfies f p = \x -> p x (f x)

soundnessTest :: [[Int]] -> Property
soundnessTest xss = product `satisfies` soundness
```

From relations to tests...

We define the corresponding QuickCheck test straight away:

```
satisfies :: (a -> b) -> (a -> b -> c) -> (a -> c)
satisfies f p = \x -> p x (f x)

soundnessTest :: [[Int]] -> Property
soundnessTest xss = product `satisfies` soundness
```

```
Main> quickCheck soundnessTest
+++ OK, passed 100 tests
```

Completeness

We might be tempted to define completeness similarly:

```
completeness :: Eq a => [[a]] -> [[a]] -> [a] -> Property
completeness xss yss xs = pairwise elem xs xss ==> elem xs yss
```

Completeness

We might be tempted to define completeness similarly:

```
completeness :: Eq a => [[a]] -> [[a]] -> [a] -> Property
completeness xss yss xs = pairwise elem xs xss ==> elem xs yss
```

But testing becomes much weaker...

```
Main> quickCheck (product `satisfies` completeness)
*** Gave up! Passed only 26 tests; 1000 discarded tests:
77% 0
19% 1
 4% 2
```

Specifying completeness

We have (very roughly) two options:

- Define a custom generator that *only* generates random lists from the cartesian product;

Specifying completeness

We have (very roughly) two options:

- Define a custom generator that *only* generates random lists from the cartesian product;
- Think harder…

Defining a custom generator

```
completeness :: (Eq a, Show a) => [[a]] -> [[a]] -> Property
completeness xss yss  =
  (any null xss && null yss)
  .||. forAll (pick xss) (\xs -> elem xs yss)
  where
    pick :: [[a]] -> Gen [a]
    pick []         = return []
    pick (xs:xss)   = (:) <$> elements xs <*> pick xss
```

Defining a custom generator

```
completeness :: (Eq a, Show a) => [[a]] -> [[a]] -> Property
completeness xss yss  =
  (any null xss && null yss)
  .||. forAll (pick xss) (\xs -> elem xs yss)
  where
    pick :: [[a]] -> Gen [a]
    pick []         = return []
    pick (xs:xss)   = (:) <$> elements xs <*> pick xss
```

But…

- The generator makes an additional case distinction, not made in our code;
- We're using quite some QuickCheck functionality – what happened to our simple spec?

Accumulating completeness

Instead of quantifying over an additional list `xs`, we accumulate all the possible choices for lists as follows:

```
completeAcc :: ([a] -> Bool) -> ([[a]] -> Bool)
completeAcc p []         = p []
completeAcc p (xs:xss)   = all (\x -> completeAcc (p . (x:)) xss) xs
```

Accumulating completeness

Instead of quantifying over an additional list `xs`, we accumulate all the possible choices for lists as follows:

```
completeAcc :: ([a] -> Bool) -> ([[a]] -> Bool)
completeAcc p []         = p []
completeAcc p (xs:xss)   = all (\x -> completeAcc (p . (x:)) xss) xs
```

To define completeness, we simply require that all such lists are an element of the output `yss`:

```
completeness :: Eq a => [[a]] -> [[a]] -> Bool
completeness xss yss = completeAcc (\xs -> elem xs yss) xss
```

Testing becomes much ‘easier’ – at the expense of writing a tricky property.

From testing to verification...

Are we still sure that the accumulating completeness property describes the right property – even if our tests pass...

From testing to verification…

Are we still sure that the accumulating completeness property describes the right property – even if our tests pass…

Perhaps this is the point to start using a proof assistant.

From tests to inductive relations

One nice property of writing a relational specification is that it is easy to turn into an *inductive relation* in a proof assistant.

Haskell:

```haskell
pairwise :: (a -> b -> Bool) -> ([a] -> [b] -> Bool)
pairwise p []       []       = True
pairwise p (x:xs)   (y:ys)   = p x y && pairwise p xs ys
pairwise p _        _        = False
```

Agda:

```agda
data Pairwise (P : A → B → Set) : List A → List B → Set where
  []   : Pairwise P [] []
  _∷_  : P x y → Pairwise P xs ys → Pairwise P (x ∷ xs) (y ∷ ys)
```

From Haskell to Agda

```
product : List (List A) → List (List A)
product xss = foldr f [[ ]] xss
  module F where
  f xs yss = foldr g [ ] xs
    module G
    g x zss = foldr (λ ys → (x : ys) ∷_) zss yss
```

As the cartesian product function only uses structural recursion (fold) and total functions - it is trivial to port to Agda.

From Haskell to Agda

```
product : List (List A) → List (List A)
product xss = foldr f [[ ]] xss
  module F where
  f xs yss = foldr g [ ] xs
    module G
    g x zss = foldr (λ ys → (x : ys) ∷ _) zss yss
```

As the cartesian product function only uses structural recursion (fold) and total functions - it is trivial to port to Agda.

But what about proving correctness properties we previously only tested?

Specification

Replacing `Bool` with `Set`, we have almost the same specification as we saw previously:

```
soundness : List (List A) → List (List A) → Set
soundness xss yss = All (λ ys → Pairwise _∈_ ys xss) yss

completeness : List (List A) → List (List A) → Set
completeness xss yss = ∀ xs → Pairwise _∈_ xs xss → xs ∈ yss
```

Specification

Replacing `Bool` with `Set`, we have almost the same specification as we saw previously:

```
soundness : List (List A) → List (List A) → Set
soundness xss yss = All (λ ys → Pairwise _∈_ ys xss) yss

completeness : List (List A) → List (List A) → Set
completeness xss yss = ∀ xs → Pairwise _∈_ xs xss → xs ∈ yss
```

Note: quantifying over all lists `xs` does not suffer the same problems as that we saw when testing using QuickCheck.

For the rest of the talk, I'll focus on completeness - the arguments for soundness are identical.

Working our way inwards

The proof is not at all hard, once you set things up right:

```
product : List (List A) → List (List A)
product xss = foldr f [[ ]] xss
...

complete : (xss : List (List A)) → completeness xss (product xss)
complete []         = ...  -- trivial
complete (xs :: xss) = f-complete xs (complete xss)

f-complete : ∀ xs → completeness xss yss → completeness (xs :: xss) (f xs yss)
```

On to the inner fold...

```
product xss = foldr f [[ ]] xss
  where
  f xs yss = foldr g [ ] xs
    where
    g x zss = foldr (λ ys → (x : ys) :_) zss yss
```

The auxiliary function is, in turn, defined using a fold.

On to the inner fold…

```
product xss = foldr f [[ ]] xss
  where
  f xs yss = foldr g [ ] xs
    where
    g x zss = foldr (λ ys → (x : ys) :_) zss yss
```

The auxiliary function is, in turn, defined using a fold.

So we need another helper lemma, establishing the completeness of g:

```
completeness xss yss →
completeness (xs : xss) zss →
completeness ((x : xs) : xss) (g xss xs yss x zss)
```

Verification as explanation

This inductive argument really helped clarify how the function worked!

More so, than only writing the QuickCheck tests.

Verification as explanation

This inductive argument really helped clarify how the function worked!

More so, than only writing the QuickCheck tests.

Exercise: for the Agda inclined reader, given that the Cartesian product function (and its correctness proof) is defined using only folds…

Define a cartesian product function that produces both the resulting list and correctness proof.

Essentially - this is the ‘tupling transformation’.

Accumulating predicate

Working in a higher-order logic such as Agda lets us formally relate the ‘accumulating’ version of completeness and the ‘direct’ definition.

Accumulating predicate

Working in a higher-order logic such as Agda lets us formally relate the ‘accumulating’ version of completeness and the ‘direct’ definition.

In fact, we show the following property holds *for any* property `P`:

```
(xss : List (List A)) →
  accumulate P xss ⇔ (∀ xs → Pairwise _∈_ xs xss → P xs)
```

Our specifications are (reassuringly) equivalent.

Conclusions

I found this a fun and instructive exercise in specification, testing and verification.

Working in a proof assistant can be very helpful: the ‘accumulating predicate’ used is not so easy to understand!

This construction generalises to arbitrary traversables – I think – for example, allowing you to turn a tree of lists into a list of trees.