On the correctness of Barron and Strachey's cartesian product function

Wouter Swierstra[1][0000−0002−0295−7944] and Jason Hemann[2][0000−0002−5405−2936]

[1] Utrecht University
w.s.swierstra@uu.nl
[2] Seton Hall University
jason.hemann@shu.edu

Abstract. How can we verify Barron and Strachey's cartesian product function? Doing so is a useful exercise in the specification, testing, and verification of a classica functional program.

Keywords: Functional programming · verification · specification · Agda

1 Introduction

Christopher Strachey is one of the founding fathers of functional programming. His lecture notes from 1963 together with David Barron, as transcribed in *Advances in Programming and Non-Numerical Computation*, contain several examples of CPL programs [1]. Among these, is a function for computing the cartesian product of a list of lists. Consider the following example call to this function, using the GHCi read-eval-print-loop:

```
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]]
```

The definition of the *product* function presented by Barron and Strachey is quite puzzling at first glance. We present it here, based on the modern definition given by Danvy and Spivey [7]:

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

This definition uses three calls to *foldr*, lexical scoping, and several higher order functions – all rather unusual for a program that is over sixty years old! This definition of Barron and Strachey's cartesian product function has been carefully dissected by Danvy and Spivey [7], who refer to it as 'possibly the first ever functional pearl.' What more could there possibly be to say about this mother of pearls?

One issue that remains unexplored is a formal correctness proof of Barron and Strachey's cartesian product function: this paper addresses precisely this issue. In doing so, we illustrate several techniques for the specification, testing and verification of functional programs, ultimately leading to an intrinsically correct function definition in the proof assistant Agda. In writing the proof, the inductive argument will elucidate the inductive structure of the function itself – giving a better insight into *how* this program computes the cartesian product.

This paper presents a mix of Haskell, Agda and REPL interactions. To distinguish between them, Haskell code is written in *italics*, Agda code is written using a sans serif font, and example REPL interactions use a `teletype` font.

2 Specification

There are numerous ways to specify a function's intended behaviour. One approach, popularised by Bird [3], is to start from an inefficient yet 'obviously correct' reference implementation. A more efficient solution may then be calculated from this specification. This approach is less suitable here for three reasons: firstly, efficiency is not our primary concern; secondly, it is not so easy to come up with a reference implementation that is both shorter and obviously correct; finally, a reference implementation can be overly restrictive, fixing the order of elements in the list that is produced. In this case, we are only concerned that all elements of the cartesian product occur in the final list. A good specification does not fix the order in which the elements occur must occur in the resulting list. This last point may seem like a theoretical issue, but pops up when Bird considers greedy algorithms or other non-deterministic computations [4, 5].

Instead, we give a *relational specification*, describing the properties of the desired outputs in terms of the function's inputs. Before we give the specification, however, we introduce a few auxiliary predicates and predicate transformers. First and foremost, we will use Haskell's *elem* function to check if a given element occurs in a list:

$$
\begin{aligned}
&\mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Bool} \\
&\mathit{elem}\ y\ [] \quad\ = \mathit{False} \\
&\mathit{elem}\ y\ (x : xs) = x \equiv y \vee \mathit{elem}\ y\ xs
\end{aligned}
$$

The definition in the standard library has been generalised to work over any arbitrary *Traversable* structure [8]; to keep things simple, however, we only consider lists. To specify the intended behaviour of the cartesian product function, we need two additional predicate transformers: *all* and *pairwise*. The *all* predicate transformer asserts that its argument predicate holds for each element of a list:

$$
\begin{aligned}
&all :: (a \to Bool) \to ([a] \to Bool) \\
&all\ p\ [] \quad\ = True \\
&all\ p\ (x : xs) = p\ x \wedge all\ p\ xs
\end{aligned}
$$

Finally, we need a second predicate transformer, *pairwise*, that asserts that a binary relation holds pairwise between two lists:

$$
\begin{aligned}
&pairwise :: (a \to b \to Bool) \to ([a] \to [b] \to Bool) \\
&pairwise\ p\ []\ \ \ \ []\ \ \ \ = True \\
&pairwise\ p\ (x : xs)\ (y : ys) = p\ x\ y \wedge pairwise\ p\ xs\ ys \\
&pairwise\ p\ \_\ \ \ \ \_\ \ \ \ = False
\end{aligned}
$$

Note that if the two lists have different lengths, they cannot satisfy any *pairwise* property.

Using all three these predicates, we define the *soundness* property that we expect of the cartesian product function:

$$
\begin{aligned}
&soundness :: Eq\ a \Rightarrow [[a]] \to [[a]] \to Bool \\
&soundness\ xss = all\ (\lambda\ ys \to pairwise\ elem\ ys\ xss)
\end{aligned}
$$

The *soundness* property states that each list in the output draws its elements from the corresponding list in the input. Without this property, we could create in the output a list whose head is drawn from the second input list. The soundness property rules out such invalid outputs. Formulated in this fashion, the property is *wholemeal* [2], defined in terms of entire lists rather than fiddling with individual indices. In particular, we never have to perform any (partial) lookup operation, but instead define a specification built exclusively from total functions.

Given this property, we use tools such as QuickCheck [6] to test whether our *product* function satisfies the specification:

$$
\begin{aligned}
&satisfies :: (a \to b) \to (a \to b \to c) \to (a \to c) \\
&satisfies\ f\ p = \lambda\ x \to p\ x\ (f\ x) \\
&soundnessTest :: Eq\ a \Rightarrow [[a]] \to Property \\
&soundnessTest\ xss = product\ `satisfies`\ soundness
\end{aligned}
$$

Unsurprisingly, QuickCheck cannot find a counterexample that falsifies this property.

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

There is one caveat: the exponential nature of the cartesian product function does create substantial outputs. To keep testing times in check, we limit the size of the random inputs that QuickCheck generates.

This specification of the cartesian product function leaves room for incorrect implementations. For example, the function that always returns an empty list, *const* [], satisfies the *soundness* property. To rule out such trivial solutions, we

also require that each list constructed by drawing arbitrary elements of the input does occur in the output list. As a QuickCheck test, we might write:

```
completeness :: Eq a ⇒ [[a]] → [[a]] → [a] → Property
completeness xss yss xs = pairwise elem xs xss ⟹ elem xs yss
```

The *completeness* property now not only refers to the input and output lists (*xss* and *yss* respectively), but also mentions a third list *xs*. The completeness statement now asserts that whenever *xs* draws its elements pairwise from the input *xss*, the list *xs* should occur in the output *yss*. If we run QuickCheck to test this property, it fails to find a counterexample again. On closer inspection, however, we have fallen victim to the 'false sense of security' that QuickCheck sometimes provides [6, §5.4.1]: although QuickCheck has not found a counterexample, it has only generated a limited number of tests. This becomes even more apparent when we classify the results according to the length of *xs*:

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

This test of completeness is not very useful! The odds of generating a random list *xs* that *happens* to occur in the cartesian product of another random list *xss* are rather slim. What to do? One solution is to write a custom generator that produces random lists in the cartesian product. Using QuickCheck's library for writing generators, we might write:

```
completeness :: (Eq a, Show a) ⇒ [[a]] → [[a]] → Property
completeness xss yss =
  (any null xss ∧ null yss)
  .||. forAll (pick xss) (λ xs → elem xs yss)
  where
    pick :: [[a]] → Gen [a]
    pick []       = return []
    pick (xs : xss) = (:) ⟨$⟩ elements xs ⟨∗⟩ pick xss
```

The random generation of an element of the cartesian product is done using the *pick* function. This repeatedly selects a random element of each list in the input *xss*. Unfortunately, the *elements* function requires a non-empty list as its argument. Hence we test a more complicated property: if the input *xss* contains an empty list element, the resulting list *yss* is empty; otherwise, each randomly generated list picked from the lists in *xss* occurs in the output *yss*.

This specification, however, is a bit more clunky: it makes an additional case distinction that our implementation does not make. What should have been a simple specification requires a custom generator and some quite sophisticated functionality from the QuickCheck library.

There is, however, yet another solution: what if we could avoid quantifying over the additional list *xs*? That would yield a specification that no longer requires a custom generator and could be tested more easily. To achieve this, we redefine the completeness property using a (higher order) accumulating parameter:

```
completeness :: Eq a ⇒ [[a]] → [[a]] → Bool
completeness xss yss = completeAcc (λ xs → elem xs yss) xss
```

Here we define *completeness* in terms of an auxiliary function, *completeAcc*. Initially, we pass a predicate stating that some list *xs* occurs in the output list *yss*. The actual work is done by the *completeAcc* function that iterates over the input list:

```
completeAcc :: ([a] → Bool) → ([[a]] → Bool)
completeAcc p []       = p []
completeAcc p (xs : xss) = all (λ x → completeAcc (p ∘ (x :)) xss) xs
```

If the input list is empty, the output list should contain the empty list. Otherwise, we assert that for each possible choice of x, drawn from the first list in our input *xs*, the completeness holds of the tail using the predicate $p \circ (x :)$. In this way, we build increasingly complex predicates to test the presence of every possible combination of elements from our input lists.

When we run the corresponding tests, QuickCheck fails to find a counterexample:

```
ghci> quickCheckWith (product `satisfies` completeness)
+++ OK, passed 100 tests.
```

Although the tests pass, this definition is more complicated than the original *soundness* property. It is not even immediately clear that the direct and accumulating versions of completeness define the same property. At this point, it makes sense to use an interactive theorem prover to check our work.

3 Formal verification

In this section, we show how to verify that the product function satisfies both the soundness and completeness properties from the previous section. To do so, we begin by redefining the boolean properties used in our specifications as inductive relations.

The binary relation _∈_ says when an element occurs in a list:

```
data _∈_ (x : A) : List A → Set where
  here  : x ∈ (x :: xs)
  there : x ∈ xs → x ∈ (y :: xs)
```

This definition is easy to read off from the Haskell definition: if the list is empty, the property is false and hence there is no corresponding constructor. If the list

is non-empty, there are two alternatives: either the element we are looking for occurs in the head, or it occurs in the tail of the list.

Similarly, the Pairwise data type lifts a relation between A and B to one over lists of A and lists of B:

```
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)
```

Again the choice of constructors mirrors the test we wrote in Haskell. Finally, inhabitants of the All data type prove that all the elements of a list satisfy a given predicate:

```
data All (P : A → Set) : List A → Set where
  []  : All P []
  _::_ : P x → All P xs → All P (x :: xs)
```

We define the specification of the cartesian product function using these predicates. The product function is correct when it is both *sound* and *complete*:

```
correctness : List (List A) → List (List A) → Set
correctness xss yss = soundness xss yss ∧ completeness xss yss
```

Just as we saw previously, the definition of soundness says that all the elements in the output list arise from picking one element from the list at the corresponding position in the input list of lists:

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

Completeness, on the other hand, quantifies over an additional list xs. Whenever this list draws its elements pairwise from the input xss, it must also occur in the output yss:

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

In contrast to the QuickCheck specification, we do not need to worry about how the lists xs are *generated*, but instead merely focus on the property we wish to assert.

3.1 Soundness

Proving soundness is fairly straightforward: we follow the inductive structure of the definition, computing the induction hypotheses we need along the way. Here we present the proof in a top down fashion:

```
sound : (xss : List (List A)) → soundness xss (product xss)
sound []          = [] :: []
sound (xs :: xss) = f-sound xs (sound xss)
```

The main soundness result establishes the base case: if our input list is empty, the All predicate used to define soundness holds trivially. If the input list is non-empty, we pass the induction hypothesis as an argument, but defer the further work to correctness of the inner loop, f-sound:

```
f-sound : ∀ xs → soundness xss yss → soundness (xs ∷ xss) (f xss xs yss)
f-sound []       ih = []
f-sound (x ∷ xs) ih = g-sound ih (f-sound xs ih)
```

From this point onwards, we will liberally use Agda's *variable* notation to quantify implicitly over unbound variables in type signatures using the usual conventions for naming lists. For instance, xss and yss are both lists of lists, whereas xs is a list. Furthermore, to refer to any locally bound function, like f, requires passing any locally bound variables (such as xss) as additional arguments. Throughout the coming lemmas, we will use the same naming convention as the definition site of the product function.

To understand the f-sound lemma, focus on the second case first: it states that the function f constructs the cartesian product of (xs ∷ xss), provided we already have yss, the cartesian product of xss. The proof itself, once again, merely constructs the induction hypothesis and defers the actual proof to the innermost fold, given by a third lemma, g-sound:

```
g-sound : soundness xss yss →
          soundness (xs ∷ xss) zss →
          soundness ((x ∷ xs) ∷ xss) (g xss xs yss x zss)
```

Once again, let's read the type first: it states that cartesian product is sound for (x ∷ xs) ∷ xss given the soundness of cartesian product for xss and for xs ∷ xss. We use yss as the name for the cartesian product of xss, and zss as the name for the cartesian product of xs ∷ xss. Although the lists of lists are quite dizzying at this point, the specification of the innermost function g is clear. Deconstructing the inductive nature of the correctness proof exposes exactly how the three nested folds relate: the outermost fold iterates over the outermost list; the second fold adds the elements of xs one by one; the innermost fold adds a single element x to the results yss and zss constructed so far. The proof of this last lemma proceeds by induction on (the soundness of) xss.

```
g-sound (ih₁ ∷ ihs₁) ih₂ = (here ∷ ih₁) ∷ g-sound ihs₁ ih₂
g-sound []           ih₂ = all-map sound-step ih₂
  where
  sound-step : Pairwise _∈_ xs (ys ∷ yss) → Pairwise _∈_ xs ((y ∷ ys) ∷ yss)
  sound-step (e ∷ es) = there e ∷ es
```

We use a few helper lemmas, essentially to show how g does not invent new elements, but rather draws them from yss or extends the intermediate results from zss with the new element x. Using the following lemma, we map over all the proofs in All P xs:

```
all-map : (∀ {x} → P x → Q x) → All P xs → All Q xs
```

3.2 Completeness

Completeness is a bit harder to establish. The pattern of induction follows the same structure as we saw previously, defining three functions that each handle an individual layer of recursion.

```
complete : (xss : List (List A)) → completeness xss (product xss)

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

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

The proofs of the first two completeness statements are (almost) identical to the soundness proofs. The completeness of g uses two helper lemmas. The first proves that if a list as occurs in xss, then a call to g with the argument a will produce a list containing a :: as:

```
g-in-xss : (xs : List A) (yss : List (List A)) →
  as ∈ xss → (a :: as) ∈ g yss xs xss a zss
g-in-xss xs yss here          = here
g-in-xss xs yss (there elem)  = there (g-in-xss xs yss elem)
```

The second lemma proves that any call to g contains all the elements in zss. This is easy to see, as the base case of g returns zss:

```
g-in-zss : (xs : List A) (xss : List (List A)) →
  as ∈ zss → as ∈ g xss xs yss x zss
g-in-zss {yss = []} xs xss       = λ a∈zs → a∈zs
g-in-zss {yss = _ :: yss} xs xss = there ∘ g-in-zss {yss = yss} xs xss
```

With these two lemmas, the proof of g-complete follows readily.

3.3 Accumulation

What about our alternative notion of completeness? In Agda, we can formulate the corresponding property easily enough:

```
accumulate : (P : List A → Set) → List (List A) → Set
accumulate P []          = P []
accumulate P (xs :: xss) = All (λ x → accumulate (P ∘ (x ::_)) xss) xs

complete-acc : List (List A) → List (List A) → Set
complete-acc xss yss = accumulate (_∈ yss) xss
```

But what about *proving* that the cartesian product function is complete? Or proving that the two definitions of completeness coincide?

The code accompanying this paper contains a direct proof that the cartesian product function satisfies this property too. More interestingly, however, we show that for *any* property, the accumulating and pairwise definitions are equivalent:

```
acc-equiv : {P : List A → Set} (xss : List (List A)) →
  accumulate P xss ⇔ ((∀ xs → Pairwise _∈_ xs xss → P xs))
```

This directly implies that the accumulating and direct definitions of completeness are equivalent. Hence we can also re-use our previous completeness proof. Proving this follows by induction, using one crucial lemma:

```
all∈ : ({x : A} → x ∈ xs → P x) ⇔ All P xs
```

The implication in each direction is used to establish the corresponding direction of acc-equiv.

4 Intrinsic verification

The inductive structure of the proof and program both coincide closely: why not do both at once the two? To achieve this, we need to abandon the simply typed folds used in Haskell, in favour of the dependently typed *induction principle* or *eliminator*:

```
elim : {P : List A → Set} →
  (∀ x {xs} → P xs → P (x ∷ xs)) →
  P [] →
  ∀ xs → P xs
elim step base []       = base
elim step base (x ∷ xs) = step x (elim step base xs)
```

Operationally, the two functions behave the same. The key difference is in the *return type*. A simply typed fold produces a value of the same type, irrespective of its input. On the other hand, the return type of the eliminator *depends* on its input value xs. As we have set out to define a function returning both a list and the proof that this resulting list is the cartesian product of its input, we need this extra generality.

To make this even more clear, we introduce a type for 'correct cartesian product', or CCP for short:

```
data CCP (xss : List (List A)) : Set where
  _,_ : (yss : List (List A)) → correctness xss yss → CCP xss
```

Such a correct cartesian product of the list xss consists of an output list yss, together with the desired correctness proof relating xss and yss. The CCP type will form the first (implicit) argument P to the eliminator, sometimes referred to as the *motive*.

We now define a correct by construction cartesian product function. The complete listing is given in Figure 4, lightly edited for the sake of legibility. Replacing elim with *foldr*, we have almost exactly the same function as in the introduction, only we now return both a list and correctness proof. We have two base cases for our proofs, but left out their definition. The type signatures of the

```
product : (xss : List (List A)) → CCP xss
product xss = elim f ([ [] ] , base) xss
  where
  base : correctness [] [ [] ]
  f : (xs : List A) → {xss : List (List A)} → CCP xss → CCP (xs :: xss)
  f xs (yss , yss-c) = elim g ([] , f-base) xs
    where
    f-base : correctness ([] :: xss) []
    g : (x : A) → {xs : List A} → CCP (xs :: xss) → CCP ((x :: xs) :: xss)
    g x (zss , zss-c) =
      (foldr (λ ys → (x :: ys) :: _) zss yss , g-correct yss-c zss-c)
```

Fig. 1. A correct-by-construction cartesian product

auxiliary functions, f and g, mention an additional (implicit) argument – but this is not used in the function's definition. The only real work – as always – is done by the innermost function, g, that uses a simple fold to construct the desired list and assembles the desired correctness proof. Written in this way, the correct by construction cartesian product function is only slightly more complicated than the original definition.

5 Discussion

In a way it is unsurprising that a function defined using a fold is (relatively) easy to test and verify. Nonetheless, the establishing the correctness of a triply nested fold, making clever use of lexical scoping, is still an amusing puzzle: finding a suitable specification, proving the required lemmata, and assembling all the pieces requires a bit of thought. In doing so, we illustrate just how far testing and verification technology has come in the last sixty years.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Barron, D.W., Strachey, C.: Programming. In: Fox, L. (ed.) Advances in Programming and Non-Numerical Computation. pp. 49–82. Pergammon Press (1966)
2. Bird, R.: A program to solve sudoku. Journal of functional programming **16**(6), 671–679 (2006)
3. Bird, R.: Pearls of functional algorithm design. Cambridge University Press (2010)
4. Bird, R., Gibbons, J.: Algorithm Design with Haskell. Cambridge University Press (2020)

5. Bird, R., Rabe, F.: How to calculate with nondeterministic functions. In: Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings 13. pp. 138–154. Springer (2019)
6. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. p. 268–279. ICFP '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266
7. Danvy, O., Spivey, M.: On Barron and Strachey's cartesian product function. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. p. 41–46. ICFP '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1291151.1291161
8. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming **18**(1), 1–13 (2008). https://doi.org/10.1017/S0956796807006326