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. This paper shows how to verify Barron and Strachey's Cartesian product function. Doing so is not only a useful exercise in the specification and verification of a classic functional program—but gives insight into the importance of formulating soundness and completeness as symmetrical properties.

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:

```
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 ]]
```

Barron and Strachey's definition of *product* is quite puzzling at first glance. We present it here, based on the modern implementation in Haskell by Danvy and Spivey [4]:

```
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! Danvy and Spivey [4] have carefully dissected Barron and Strachey's Cartesian product function, and 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 mechanized formal correctness proof of Barron and Strachey's Cartesian product function. Mechanization would certainly have been out of reach for those authors back in 1963—four years before even the start of the Automath project [2, 8]. We believe there is still mystery to be wrung out of this definition.

In writing our proofs, the inductive argument will elucidate the inductive structure of the functions themselves—giving a better insight into *how* this function computes a Cartesian product.

Throughout the remainder of this paper, we will use the programming language and proof assistant Agda [9] to specify and verify the Cartesian product function. The key technique, inspired by Danielsson's work on bag equivalence [3], is to verify that the desired soundness and completeness properties not only hold, but form an *isomorphism*, establishing a much stronger result. The underlying technique for specifications and proofs—establishing an isomorphism between two symmetric components of the specification—deserves to be more widely known. More generally, this technique illustrates the importance of proof relevance, where we are interested not only in whether or not a given statement is true, but also in accounting for the different possible proofs. The code associated with this paper has been published separately [10].

2 Warm-up: permutations

Rather than study the Cartesian product function immediately, we begin by a slightly simpler case study: computing all the permutations of a list. This is a typical exercise in functional programming; the verification of this function will introduce some of the scaffolding and specifications used in the remainder of the paper.

The permute function computes all the possible permutations of its input list:

```
permute : List A → List (List A)
permute []       = [ [] ]
permute (x :: xs) = concatMap (sprinkle x) (permute xs)
```

The work is done by the sprinkle function that sprinkles its first argument in all the possible positions in its second argument:

```
sprinkle : A → List A → List (List A)
sprinkle y []       = [ [ y ] ]
sprinkle y (x :: xs) = (y :: x :: xs) :: map (x :: _) (sprinkle y xs)
```

So far so good—but we still need state what 'correctness' means for computing permutations and to prove that our function satisfies this specification. In the case of permutations, we are interested in two separate properties:

- *soundness* states that all the elements of the resulting list are permutations of the input;
- *completeness* states that every possible permutation of the input appears in the output.

The problem with writing such specifications in English is that we have cleverly side-stepped the problem of specifying when one list is a permutation of another—which is not necessarily any easier than the problem we set out to solve in the first place!

There are many different ways to specify that two lists are permutations of one another: by repeatedly swapping adjacent elements (as is done in the Rocq standard library); by quotienting the laws of commutative monoids [5]; by counting the number of occurrences of each element in the list; or, as we shall see shortly, by establishing an isomorphism between membership *proofs*.

To start our exploration of list processing functions, we begin with defining the binary relation, written $\mathsf{x} \in \mathsf{xs}$, that states that an element x occurs in a list xs:

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

This definition is reminiscent of the Haskell *elem* function: 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 is the first element, here, or it occurs in the tail of the list, there.

Besides this relation, our specification will use an additional *predicate transformer* on lists, given by the All data type below:

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

The proposition All P xs holds precisely when we have a proof of P x for each element x of of xs.

As a first approximation, we might define the following types to formalize our notions of soundness and completeness for our permute function:

```
soundness : (xs : List A) →
  ∀ ys → ys ∈ permute xs → All (_∈ xs) ys
completeness : (xs : List A) →
  ∀ ys → All (_∈ xs) ys → ys ∈ permute xs
```

This specification, however, is not very good. In words, the soundness property states that every list ys in permute xs draws its elements from xs. As it is stated, however, this property fails to prohibit illegal 'permutations' that might duplicate elements or discard elements. To illustrate this point, consider the following lists:

```
one-one  = 1 :: []
two-ones = 1 :: 1 :: []
```

Now both All (_∈ one-one) two-ones and All (_∈ two-ones) one-one hold—yet clearly the lists are not permutations of one another. The completeness property is similarly problematic: just because the list ys contains all the elements in xs does not make ys a permutation of xs. Clearly we need to account for the *number* of occurrences of each element more carefully. In a way, the soundness and completeness properties as formulated above consider lists as *sets* rather than *bags*.

To address this problem, we take inspiration from Danielsson's work on defining bag equivalence using a proof relevant membership relation [3]. The key idea is to consider two lists to be bag equivalent if there is an *isomorphism* between their membership relations. To make this more precise, we begin by defining the notion of isomorphism:

```
record _≃_ (A B : Set) : Set where
  field
    to     : A → B
    fro    : B → A
    to-fro : ∀ x → fro (to x) ≡ x
    fro-to : ∀ y → to (fro y) ≡ y
```

Two lists are bag equivalent if there is an isomorphism between membership proofs. We formalise this relation as follows:

```
_≈_ : List A → List A → Set
xs ≈ ys = ∀ x →  x ∈ xs  ≃  x ∈ ys
```

In words, two lists xs and ys are bag equivalent, written xs ≈ ys, when there is a one-to-one correspondence between the proof of x ∈ xs and x ∈ ys, for every x. It is straightforward to prove that this definition of bag equivalence does form an equivalence relation.

A better specification for our permutation function now states:

```
soundness : (xs : List A) →
  ∀ ys  →  ys ∈ permute xs  →  xs ≈ ys
completeness : (xs : List A) →
  ∀ ys  →  xs ≈ ys  →  ys ∈ permute xs
```

We will now briefly sketch the key lemmas used to prove these two properties of the permute function.

Soundness The proof of soundness relies on the property that sprinkle 'preserves' bag equivalence:

```
sprinkle-equiv : ys ∈ sprinkle x xs  →  ys ≈ (x :: xs)
```

More precisely, we should say that each list ys in the list of lists produced by a call to sprinkle x xs is bag equivalent to x :: xs.

To establish soundness, however, needs a bit more work. In particular, it is not immediately obvious how to use an assumption of the form:

```
ys ∈ concatMap (sprinkle x) xs
```

To do so, we prove an equivalence between the elements of a list we generated by a call to concatMap:

```
(y : B) (f : A → List B) (xs : List A) →
  y ∈ concatMap f xs  ≃  ∃ [ x : A ] (x ∈ xs  ∧  y ∈ f x)
```

Here we use the logical conjunction symbol, ∧, to create a pair of proofs; the prevalent notation A × B is confusing when the types A and B also contain the variable x. In our implementation, we have packaged the fact of the concatMap membership equivalence into a custom *view* [7], letting us tease apart our assumption more easily. The proof of soundness follows from these ingredients using a straightforward inductive argument.

Completeness To prove completeness is a bit trickier. In particular, our assumption that some list ys is a permutation of xs is itself not an inductively defined relation. Instead, we use the isomorphism between the elements of a concatMap in the other direction. Doing so, relies on a corresponding completeness result for sprinkle:

```
sprinkle-complete : (p : x ∈ xs)  →  xs ∈ sprinkle x (remove xs p)
```

This proves that the sprinkle at least recovers the original list xs, after removing an arbitrary element from it. Here the remove function deletes the occurrence referenced by the proof p:

```
remove : (xs : List A) → x ∈ xs → List A
remove (x :: xs) here      =  xs
remove (x :: xs) (there i) =  x :: remove i
```

To prove completeness, we perform induction on the list ys. When this list is empty, our hypothesis tells us that xs must also be empty, and the proof is trivial. In the inductive case, however, we need to do a bit more work. We use the concatMap-isomorphism from the previous section in the opposite direction. The sprinkle-complete lemma and our induction hypothesis finish the proof. To use our inductive hypothesis, however, we need to establish that if x :: xs ≈ ys, then xs is bag equivalent to removing some particular element of ys, which we leave as a finicky proof for the reader.

Correctness Even if we establish soundness and completeness, this is *still* not sufficient to guarantee the correctness of our permute function. There is nothing ruling out the following (incorrect) variation of the permute function:

```
wrong-permute : List A → List (List A)
wrong-permute xs = permute xs ++ permute xs
```

Indeed, we should prove that the soundness and completeness functions are mutual inverses, which involves equivalences of (bag) equivalences —the mind boggles.... Rather than go down this particular homotopical rabbit hole, however, we now return to our original problem: proving the correctness of the Cartesian product function.

3 Cartesian products

We now turn our attention to the verification of the Cartesian product function from the introduction. We verify the function in two steps. First, we show how the version by Barron and Strachey is equivalent to a simpler definition using concatMap. Next, we establish the correctness of this second version. Arguably, this style of proof introduces an unnecessary indirection; we briefly sketch the direct proof in Section 4.

3.1 From concatMap to nested folds

Barron and Strachey's definition of Cartesian product seems perplexing at first and what's more, as Danvy and Spivey write, the authors seem almost to pull this startling definition out of thin air. To better explain how the triply nested fold arises, we have formalised a derivation of this function, suggested by a reviewer, starting from a simpler definition. The proof sketch of this derivation is in Figure 1.

The following reference implementation of the Cartesian product function is a bit easier to understand and corresponds more closely to what one might write initially:

```
product-spec : List (List A) → List (List A)
product-spec []       = [ [] ]
product-spec (xs :: xss) =
  concatMap (λ x → concatMap (λ ys → [ x :: ys ]) (product-spec xss)) xs
```

The derivation in Figure 1 illustrates the inductive case, establishing that this reference implementation is the same as the one from Barron and Strachey given in our introduction. After applying the induction hypothesis, the next two steps replace the calls to concatMap with the corresponding fold, using the following identity:

```
concatMap f xs ≡ foldr (λ x ys → f x ++ ys) [] xs
```

The penultimate step relies on the following property, essentially allowing us to fuse a call to map and append into a single fold:

```
foldr (λ x ys → f x :: ys) [] xs ++ zs ≡ foldr (λ x ys → f x :: ys) zs xs
```

```
product-spec (xs ∷ xss)
  ≡⟨  definition of product-spec  ⟩
concatMap (λ x → concatMap (λ ys → [ x ∷ ys ]) (product-spec xss)) xs
  ≡⟨  induction hypothesis  ⟩
concatMap (λ x → concatMap (λ ys → [ x ∷ ys ]) (product xss)) xs
  ≡⟨  concatMap as foldr  ⟩
foldr (λ x zss → concatMap (λ ys → [ x ∷ ys ]) (product xss) ++ zss) [] xs
  ≡⟨  concatMap as foldr  ⟩
foldr (λ x zss → foldr (λ ys qss → (x ∷ ys) ∷ qss) [] (product xss) ++ zss) [] xs
  ≡⟨  property of foldr  ⟩
foldr (λ x zss → foldr (λ ys qss → (x ∷ ys) ∷ qss) zss (product xss)) [] xs
  ≡⟨  definition of product  ⟩
product (xs ∷ xss)
□
```

Fig. 1. Derivation of Barron and Strachey's Cartesian product function

This is used to replace the empty list in the base case with the previously computed elements of the Cartesian product, zss. To achieve the definition from the introduction, we would need to introduce the auxiliary functions, f and g, explicitly; Agda happily unfolds these definitions, however, completing the proof for us. The complete listing of the Agda reimplementation of Barron and Strachey's function is in Figure 2.

3.2 Verification

With this simpler definition in place, we finally give the specification and correctness proof of the Cartesian product function. To specify the desired behaviour, we define another predicate transformer. 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)
```

The Pairwise predicate transformer is similar to Haskell's zipWith function, requiring a proof of P x y for all elements x and y that occur at the same position in xs and ys respectively. Note that, unlike zipWith, the Pairwise predicate enforces that both lists have *exactly* the same length. There is no possible proof of a pairwise predicate if the argument lists have different lengths.

Just as we saw in the previous section, we formulate both *soundness* and *completeness* properties:

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

Both of these properties are slightly more general than necessary: rather than mention product xss, we refer to an arbitrary list yss. Instantiating yss to the list produced by product xss gives the soundness and correctness properties you might expect. The soundness property then states that all the elements listed by the product function must draw their elements from the input lists; the completeness property then states that any list that does so must be in the list computed by the product function. By generalizing over the list yss, however, we can re-use these properties to talk about the soundness and completeness of other functions—as we shall see in the next section.

Inspired by the bag equivalence relation from the previous section, we now formulate the overall correctness result we wish to establish:

```
correctness : List (List A) → Set
correctness xss = ∀ ys → ys ∈ product-spec xss ≃ Pairwise _∈_ ys xss
```

The soundness and completeness results each form one of the functions in this isomorphism. This overall correctness result ensures that the two functions are each other's inverse. This stronger property ensures that the Cartesian product function does not duplicate or discard elements: there is always a unique proof that each element in the Cartesian product draws its elements from the input lists.

Soundness The soundness proof is somewhat involved. Given the assumption that ys ∈ product-spec xss, we need to establish that ys draws its elements pairwise from xss. In the inductive case, we deconstruct our assumption into several smaller parts:

```
ys ∈ concatMap f (product-spec xss) →
  ∃ [ xs : List A ] ((xs ∈ product-spec xss) ∧ (ys ∈ f xs))
```

Using this lemma twice, we produce the desired proof that the head of ys occurs in the head of xss; the inductive hypothesis then completes the proof.

Completeness Somewhat surprisingly, the proof of completeness is almost trivial. We perform induction on our assumption that ys draws its elements pairwise from product-spec xss. In the inductive case, we have all the ingredients necessary to establish that ys ∈ product-spec (xs ∷ xss). We use the lemma from the previous section to establish that ys is an element of a list in the image of two calls to concatMap.

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

Fig. 2. An Agda implementation of Barron and Strachey's Cartesian product function.

Correctness The overall correctness proof establishes that the soundness and completeness proofs are mutual inverses. The proof itself is tedious, but technical, following the inductive structure of both lemmas.

The overall strategy for the proof in this section has been relate the Cartesian product from §1 to a simpler reference implementation, followed by the verification of this reference implementation. In the next section we show how to avoid this indirection and give a direct correctness proof Barron and Strachey's original implementation.

4 A direct proof

To prove the soundness and completeness of the original implementation, we begin by giving the complete definition in Agda in Figure 2. A technical detail worth mentioning is that the additional local modules F and G enable us to refer to the locally defined functions f and g. These functions also generalize over all local variables in scope, in their order of occurrence. For example, the function g takes arguments xss, xs, yss, x and zss in that order; the function f takes arguments xss, xs and yss, even though it does not directly use xss.

4.1 Soundness

To establish the soundness of the original Cartesian product function, we need to prove:

```
soundness-direct : (xss : List (List A)) → soundness xss (product xss)
```

The base case follows readily enough; the inductive step requires an auxiliary lemma about the auxiliary function f:

```
f-soundness : ∀ xs → soundness xss yss → soundness (xs ∷ xss) (f xss xs yss)
```

The corresponding proof defers all the work to a final lemma about the function g:

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

This lemma makes the inductive nature of the nested-fold explicit: the outermost fold traverses the outermost list of lists; the auxiliary function f adds a new list xs to this list of lists; and finally, the function g adds elements the inner list xs. The proof itself requires two lemmas:

```
sound-there : Pairwise _∈_ xs (ys :: yss) → Pairwise _∈_ xs ((y :: ys) :: yss)
elem-destruct : ys ∈ foldr (λ zs → _::_ (x :: zs)) yss xss →
  Either (∃ [xs : List A] (ys ≡ x :: xs  ∧  xs ∈ xss))
         (ys ∈ yss)
```

The first lemma is used to extend the inner list with new elements. The second is used to decompose the hypothesis that ys occurs in the fused map-fold: either ys is in the image of map (x ::_) or it occurs in the list yss, returned when xss is empty.

4.2 Completeness

The completeness proof is simpler, following exactly the same inductive structure. The types of the three key lemmas follow the same pattern as we saw for soundness:

```
product-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)
```

Once again, the only work is done by the last lemma about the auxiliary function g. This lemma traverses its argument Pairwise proof, using the following properties of g:

```
g-in-xss : ys ∈ xss → (y :: ys) ∈ g yss xs xss y zss
g-in-zss : ys ∈ zss → ys ∈ g xss xs yss x zss
```

Essentially, both these properties illustrate how the elements produced by g are either in the remainder of the Cartesian product (zss) or produced by adding the current head of the first list, y, to another list. The proofs of both these properties follow by a simple inductive argument.

5 Intrinsic verification

Now that the inductive structure of the proof and program both coincide closely: why not do both at once? To achieve this, we need to abandon the simply typed folds used in Haskell, in favor 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 functions foldr and elim 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* [6].

We now define a correct by construction Cartesian product function. The complete listing is given in Figure 3, lightly edited for the sake of legibility. Replacing elim with foldr yields almost exactly the same function as the one in the introduction, only we now return both a list and a correctness proof. We have two base cases for our proofs, base and f-base, but we omit their definitions. The type signatures of the 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—phrased in this way, we do not need the extra generality of the dependently typed elim function as we merely compute a list. The proof component is built by the function g-correct, that calls the soundness and completeness results from the previous section. Written in this way, the correct by construction Cartesian product function is only slightly more complicated than the original definition.

```
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. 3. A correct-by-construction Cartesian product

6 Discussion

It is unsurprising that a pure function defined using a fold is (relatively) easy to test and verify. Nonetheless, establishing the correctness of a triply nested fold that makes clever use of lexical scoping is still an amusing puzzle: finding a suitable specification, proving the required lemmata, and assembling the pieces all require a bit of thought. In particular, the symmetry in our specifications enables us to express the isomorphism of proofs succinctly.

This symmetry between the soundness and completeness is no accident. We have tried several other equivalent formulations. Although these alternative formulations are logically equivalent, they make it harder to express the required property that soundness and completeness are mutual inverses.

All soundness Rather than express soundness by mentioning individual elements of the output list, we can give an alternative, capturing the desired characteristic property of all the results returned by the Cartesian product function:

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

This definition is equivalent to the one introduced previously in Section 3. This follows readily from the following property, relating individual elements and the All predicate:

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

Accumulating completeness The definition above shows how to specify soundness without mentioning individual elements. In the quest for symmetry, we have

also considered alternative notions of completeness. Defining a suitable notion of completeness in this style is not entirely straightforward. One way to do so is using an *accumulating parameter*:

```
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
```

This avoids the additional quantification of the list ys. Like the notion of soundness above, this notion of completeness is described in terms of All. Once again, this definition is logically equivalent to the one we have seen previously. *A fortiori*, we show that for any predicate P the accumulating and pairwise specifications 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—but the formulation used in the paper makes it easier to express the isomorphism. The required proofs mirror the inductive structure of the underlying definitions. The proof from left to right is defined tail-recursively, extending the predicate P as we recurse over the input list and pairwise proofs. The other direction follows by induction on the input list, using the following lemma to accumulate the elements of the input list accordingly:

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

Conclusion Even setting aside the challenges of proving correspondence to an implementation, writing good specifications is hard. A specification that is too loose leaves room for incorrect implementations, as we saw with the wrong-permute function. Conversely, one that is too strict rules out valid programs, for example, by fixing the order in which the elements of a Cartesian product function must be generated. Expressing soundness and completeness symmetrically, however, has paid dividends: we establish a one-to-one correspondence between the elements of our Cartesian product and the proofs that these are valid.

Acknowledgments. We would like to thank the reviewers and attendees of TFP for their helpful feedback and suggestions. They have helped improve the paper enormously. Nicolas Wu suggested studying permutations before moving on to the Cartesian product function itself. The research of Jason Hemann has been partially supported by NSF grant CCF-2348408.

Disclosure of Interests. The authors have 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. de Bruijn, N.G.: Automath: A language for mathematics. EUT report 68-WSK-05, Technische Hogeschool Eindhoven (1968), https://pure.tue.nl/ws/portalfiles/portal/2039924/256169.pdf, wSK, Dept. of Mathematics and Computing Science
3. Danielsson, N.A.: Bag equivalence via a proof-relevant membership relation. In: Interactive Theorem Proving: Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings 3. pp. 149–165. Springer (2012)
4. 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 (2007). https://doi.org/10.1145/1291151.1291161
5. Dinges, A., Hinze, R.: What's in a bag?: An "Application Proving Interface" for finite bags and its implementation. In: Proceedings of the 35th Symposium on Implementation and Application of Functional Languages. IFL '23, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3652561.3652563
6. McBride, C.: Elimination with a motive. In: International Workshop on Types for Proofs and Programs. pp. 197–216. Springer (2000)
7. McBride, C., McKinna, J.: The view from the left. Journal of functional programming **14**(1), 69–111 (2004)
8. Nederpelt, R.P., Geuvers, J.H., de Vrijer, R.C.: Selected Papers on Automath. Studies in Logic and the Foundations of Mathematics, Elsevier, Netherlands (1994). https://doi.org/10.1016/s0049-237x(08)70226-x
9. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
10. Swierstra, W., Hemann, J.: Proofs associated with the paper *'On the Correctness of Barron and Strachey's Cartesian Product Function'* (Apr 2025). https://doi.org/10.5281/zenodo.15118185