# Cartesian Stair Pair Combinatorics

by Sven Nilsen, 2020

*In this paper I present a combinatorics over Cartesian products using stair pairs as generators.*

A unit element is a Cartesian product[1] over zero arguments:

| | | |
|---|---|---|
| () | unit element | 0 |

For readability, I will use \`() = 0\`.

A singleton is a Cartesian product over a single argument:

| | | | |
|---|---|---|---|
| (()) | singleton | (0) | 1 |

For readability, I will use \`(()) = 1\`.

A singleton can be constructed for any Cartesian product.
Notice that a singleton construction is isomorphic to the successor function of natural numbers[2].
In this combinatorics, such constructions are ignored and represented by \`(())\`.
This choice makes it possible to use natural numbers for other structures:

2 = (1, 1)
3 = (1, 1, 1)
4 = (1, 1, 1, 1)
n = (1, 1, …)

For every \`n : nat\`, the Cartesian product of \`n\` singletons is labeled \`n\`.

Naturally, this holds for the unit element \`()\`.

This holds also for the choice of ignoring successors:

((())) = (1) = 1

Now, the labels will be reused such that for any \`a\` and \`b\` with a leftover,
they can be contracted to \`a + b\`:

| | |
|---|---|
| 2 = (1, 1) = (2) | not valid since there is no leftover |
| 3 = (1, 1, 1) = (1, 2) = (2, 1) | valid since there is a leftover |

Another way to formulate this law, is to “lift up” any element generated by a label:

3 = (2, 1) = ((1, 1), 1) = (1, 1, 1)

One can think about a label as a grammar for Cartesian products.

To generate the grammar for Cartesian products, one can use stair pairs recursively.
Stair pairs can be used to enumerate all pairs `(a, b)` where the sum is constant[3].

A stair pair has the property that for every pair, there is an associated number:

| | |
|---|---|
| (a, b) | a + b – 1 |

By adding one to this number, one gets the sum `a + b`.

Since the labels `() = 0` and `(()) = 1` can not be represented as pairs,
and `2 = (1, 1)` does not satisfy `a < b`, the pairs must start at `3`.
This means that one must add `3` to the associated number, which is `a + b + 2`.
To fix the sum, one adds `1` to each component of the pair:

| | |
|---|---|
| (a + 1, b + 1) | a + b + 2 |

A simpler way to do this is to use an asymmetric path and solve it:

```
∴       stair_pair[(+ 3) → ((+ 1), (+ 1))](n : nat) =
            if even(n) {(n / 2 – 1, n / 2 + 1)}
            else {((n – 1) / 2, (n – 1) / 2 + 1)}

∵       stair_pair(n : nat) = if even(n) {((n + 2) / 2 – 1, (n + 2) / 2)}
                                 else {((n + 3) / 2 – 2, (n + 3) / 2)}
```

When adding `3`, an even number becomes odd and vice versa, so the if-cases are swapped.

The remaining work is the following:

- All swaps `(b, a)` for `(a, b)`
- If the label `n` is even and `n > 0`, then add grammars of `(n / 2, n / 2)`

When lifting is not included, this generates every binary Cartesian product exactly once.
Such Cartesian products can be generated without any overhead.

For each lift, one can use a hash table to check whether an element has been generated.
One can use the check on grammars, to avoid expanding an entire branch if it already was visited.

This introduces some overhead, but is relatively easy to program instead of dealing with every case.
As a result, every Cartesian product can be enumerated, ignoring successor singletons.

## References:

[1] “Cartesian product”
Wikipedia
https://en.wikipedia.org/wiki/Cartesian_product

[2] “Peano axioms”
Wikipedia
https://en.wikipedia.org/wiki/Peano_axioms

[3] “Stair Pairs”
Sven Nilsen, 2020
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/stair-pairs.pdf