

# Cross Argument Asymmetric Path Notation

by Sven Nilsen, 2017

Sometimes it is useful to convert functions to a different form. For example, one argument can be split into two, two arguments can be merged into one, or two arguments can be swapped.

Cross argument asymmetric path notation allows two arguments of type  $b \times c$  become merged into a type  $bc$ . This is possible because it is the natural way of interpreting a path when the types do not match one-to-one:

$$f: a \times b \times c \rightarrow a$$

$$g_0: a \rightarrow a$$

$$g_1: b \times c \rightarrow bc$$

$$g_2: a \rightarrow a$$

$$f[g_0 \times g_1 \rightarrow g_2]: a \times bc \rightarrow a$$

In Nilsen cartesian product notation, you can infer the types of the function product as long only merging is used:

$$f: a \times b \times c \rightarrow a$$

$$f[g_{i \rightarrow n}]: a \times bc \rightarrow a$$

$$g_{in}: (a \rightarrow a, b \times c \rightarrow bc, a \rightarrow a)$$

In the case of splitting, you must use parentheses in the path type:

$$f: a \times bc \rightarrow a$$

$$f[g_{i \rightarrow n}]: a \times (b \times c) \rightarrow a$$

$$g_{in}: (a \rightarrow a, bc \rightarrow b \times c, a \rightarrow a)$$

This is because a function always returns a single type, not an associative product type. It removes the ambiguity when the argument types of the path is read in a different way:

$$f: a \times bc \rightarrow a$$

$$f[g_{i \rightarrow n}]: (a \times b) \times c \rightarrow a$$

$$g_{in}: (a \rightarrow a \times b, bc \rightarrow c, a \rightarrow a)$$

Therefore, the number of argument in the path must be less or equal than number of arguments of the function. This is also consistent with the idea that a path always belongs to an equal or less function space, and also makes binary functions naturally the largest function spaces.

To split an argument into two, you need to define a replacement. Often the implementation can be automatically derived:

$$f: a \times (b \times c) \rightarrow a$$
$$h: a \times b \times c \rightarrow a$$
$$h[id \times \text{pair} \rightarrow id] \leq f$$
$$h = \lambda(a, b, c) = f(a, (b, c))$$

Nothing prevents you from implementing a function by calling another function from a larger function space.

The function ``pair`` is defined with currying to avoid ambiguity with ``id``:

$$\begin{aligned} \text{pair} &: a \rightarrow b \rightarrow a \times b \\ \text{pair } x \ y &= (x, y) \end{aligned}$$

To swap two arguments, you can use a ``swap`` function and then a split:

$$f: a \times b \times c \rightarrow a$$
$$f[id \times \text{swap} \rightarrow id]: a \times (c \times b) \rightarrow a$$
$$h[id \times \text{pair} \rightarrow id] \leq f[id \times \text{swap} \rightarrow id]$$
$$h: a \times c \times b \rightarrow a$$

The swap function is defined with currying to avoid ambiguity:

$$\begin{aligned} \text{swap} &: a \rightarrow b \rightarrow b \times a \\ \text{swap } x \ y &= (y, x) \end{aligned}$$

In practice, even if swapping is possible using cross argument asymmetric paths notation alone, you probably want to use a higher order function:

$$f: a \times b \times c \rightarrow a$$

```
// Swap the first and the third argument.  
swap_arg(0, 2, f): c × b × a → a
```