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 × c` become merged into a type `bc`. This is possible because it is the natural way of interpreting a path when the types do no match one-to-one:

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

$$g_0 : a \to a$$

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

$$g_2 : a \to a$$

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

In Path Function Product Notation[1], you can infer the types of the function product as long only merging or swapping is used:

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

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

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

In the case of swapping:

$$f : a \times b \times b \to a$$

$$f[g_{i \to n}] : a \times b \times b \to a$$

$$g_{in} : (a \to a,\ b \times b \to b \times b,\ a \to a)$$

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

$f: a \times bc \to a$

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

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

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:

$f: a \times bc \to a$

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

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

Therefore, the number of argument in the path must be less or equal than number of arguments of the function (merge or swap). 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. This might require special rules. Often the implementation can be automatically derived:

$f: a \times (b \times c) \to a$

$h: a \times b \times c \to a$

$h[id \times pair \to id] <=> f$

$h = \backslash(a, b, c) = f(a, (b, c))$

Nothing prevents you from implementing a function by calling another function from a larger function space. If the implementation can be automatically derived, these functions always exist.

Defining the function `pair` with currying to avoid ambiguity with `id`:

```
pair : a → b → a × b
pair x y = (x, y)
```

The special rule is to feed in the next argument when types do not match by default.

Alternative is to define `pair` this way:

$pair_{a,b} <=> id_{(a, b)}$

References:

[1] “Path Function Product Notation”
Sven Nilsen, 2017-2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/path-function-product-notation.pdf