Cocategory Enumeration

by Sven Nilsen, 2020

In this paper I suggest a technique for efficient enumeration based on paths as a cocategory.

In function application[1], the value of the expression is the output:

    f(a) = b

In a path sub-type[2], the value of the expression is the possible inputs.

    a : [f] b

In path semantics[3], one might think of function application as lifting the input value to the type level, but keeping the output. The path "lifts" the output to the type level, but keeps the input.

Therefore, one can use the following intuition:

path ~= application Paths are morphisms in a cocategory[4] of Set, written Set^{op}

Here, \`~=\` means there exists a parameterized functor[5] from \`application\` to \`path\`
and a forgetful functor[6] from \`path\` to \`application\`.

Cocategory Enumeration is a technique for efficient enumeration based on \`path ~= application\`.

For every function application, there is a corresponding analogue of a path.
Therefore, one expects operations, such as products or partial application to carry over.
The operations that corresponds to operations in \`Set\` are higher operations on iterators in \`$\text{Set}^{op}$\`.

The local domains[7] for a parameterized set of outputs are non-overlapping (for pure functions).
This means that when enumerating over the local domains, their iterators can be concatenated.

For example, if \`[f] a\` and \`[f] b\` constructs two iterators,
then these two iterators can be concatenated without overlap when \`a ¬= b\`.
When composing paths, one can use a path iterator to parameterize the concatenation.
The sub-type \`[f] [g] a\` is simplified to \`[f] b\` where \`b\` is parameterized by \`[g] a\`.

For example, in Rust[8], the trait for iteration is \`Iterator\`. \`IntoIterator\` generates an iterator.
Since iterators are parameterized in cocategory enumeration,
one can use a new \`HigherIntoIterator\` trait, such that for \`[F] Y\`, \`F : HigherIntoIterator<Y>\`:

```
pub trait HigherIntoIterator<T> {
    type Item;
    type IntoIter;
    fn hinto_iter(self, arg: T) → Self::IntoIter;
}
```

References:

[1] "Function application"
Wikipedia
https://en.wikipedia.org/wiki/Function_application

[2] "Sub-Types as Contextual Notation"
Sven Nilsen, 2018

[3] "Path Semantics"
AdvancedResearch
https://github.com/advancedresearch/path_semantics

[4] "cocategory"
nLab
https://ncatlab.org/nlab/show/cocategory

[5] "Functor"
Wikipedia
https://en.wikipedia.org/wiki/Functor

[6] "Forgetful functor"
Wikipedia
https://en.wikipedia.org/wiki/Forgetful_functor

[7] "Domain of a function"
Wikipedia
https://en.wikipedia.org/wiki/Domain_of_a_function

[8] "Rust"
The Rust programming language
https://www.rust-lang.org/