Path Sets

by Sven Nilsen, 2017

Sometimes a path[1] is not unique, but has multiple solutions:

f[g] <=> {h_0, h_1, …}

This happens when the path is non-surjective[2], for example:

id[false_1] <=> {false_1, id}

A non-surjective path collapses the domain[3] of `h`, such that it becomes partial[4]. All functions that contain the same partial function is member of the path set.

A path can also be non-existent, which corresponds to an empty set. This gives paths a natural semantics that corresponds to set theory.

If one thinks of variables as functions with 0 arguments, then it is easy to generalize function application[5] to operate on sets of functions, where the result is also a set of functions:

not({false}) <=> {true}

not({false, true}) <=> {true, false}

This makes it possible to do function currying[6] in a way that corresponds to paths:

and({true}) <=> {id}

and({false}) <=> {false_1}

and({false, true}) <=> {false_1, id}

and({false, true}) <=> id[false_1]

Actually, `id[false_1]` is not the only path:

id[false_1] <=> {false_1, not, id, true_1}[false_1]

and({false, true}) <=> {false_1, not, id, true_1}[false_1]

Instead of writing every function of type `bool → bool`, one can just write:

and({false, true}) <=> (bool → bool)[false_1]

This is only allowed because all functions in the set have the same path set by `false_1`. You take the intersection of the sets for all functions in `bool → bool`.

The set of all functions of type `bool → bool` have no partial function in common, so you can not write:

```
f[g] <=> (bool → bool)
```

This is because there is no function you can construct with a path such that it is logically equivalent to all functions of type `bool → bool`. The type `bool` is non-empty, so the function space `bool → bool` is non-empty. The only case where you can do this is when the function space is empty.

The function space `bool → bool` has these properties:

```
(bool → bool)[false_1] <=> {false_1, id}
(bool → bool)[not] <=> {}
(bool → bool)[id] <=> {}
(bool → bool)[true_1] <=> {id, true_1}
```

You can write sets of paths that have the same path sets:

```
(bool → bool)[{not, id}] <=> {}
```

If path sets are not the same for a set of paths, then you take the intersection of the path sets:

```
(bool → bool)[(bool → bool)] <=> {}
```

Now, we construct a higher order function logically equivalent to `if`:

```
if : a → a → (bool → a)
if = \(x, y) = \(c) = if c {x} else {y}
```

Since `and` can be curried with `true` and `false`, we can construct `and` using `if`:

```
and(true) <=> id
and(false) <=> false_1

if(id, false_1) <=> and
```

Doing the same for `or`:

```
or(true) <=> true_1
or(false) <=> id

if(true_1, id) <=> or
```

It is easy to see that one can derive the symmetric paths of `if` by `not`:

```
and[not] <=> or
if(id, false_1)[not] <=> and[not]
if(id, false_1)[not] <=> if(true_1, id)
```

In general, the only interesting path by `bool → bool` is `not`, because the other 3 are trivial:

> if(a, b)[false_1] <=> false_n
> if(a, b)[not] <=> if(b[not], a[not])
> if(a, b)[id] <=> if(a, b)
> if(a, b)[true_1] <=> true_n

Any boolean function can be constructed by `if`. Its symmetric path by `not` can be simplified:

> if(if(a_0, a_1), if(a_2, a_3))[not]
> if(if(a_2, a_3)[not], if(a_0, a_1)[not])
> if(if(a_3[not], a_2[not]), if(a_1[not], a_0[not]))
> [not] if(if(a_3, a_2), if(a_1, a_0))
> not(if(if(a_3, a_2), if(a_1, a_0)))
>
> a_i : bool

Notice that it just reverses the order and inverts the output. You can easily do the same to any else-if expression by copying the expression, reversing the higher order arguments and inverting the output:

> f := \(a_0, a_1, a_2) = \(x_0, x_1) = if x_0 { a_0 } else if x_1 { a_1 } else { a_2 }
>
> f[not] := \(a_2, a_1, a_0) = \(x_0, x_1) = !if x_0 { a_0 } else if x_1 { a_1 } else { a_2 }
>
> a_i : bool

A summary so far:

1. Paths form a set called "path set"
2. All boolean functions and their symmetric paths can be constructed with `if`

Now we want to study the cardinality sum of all symmetric and asymmetric path sets.

$$\sum j \{ |f_j[g_{i\to n}]| \}$$

Creating a table for the function space `bool → bool`:

Column → Row	**false_1**	**not**	**id**	**true_1**
false_1	8	4	4	8
not	4	4	4	4
id	4	4	4	4
true_1	8	4	4	8

$$\sum j \{ |f_j[g_{i\to n}]| \} = 4^2\cdot 4 + 4\cdot(8-4) = 4\cdot 4\cdot 4 + 4\cdot 4 = 80$$

This was counted manually. In the future I might be able to generalize this further.

References:

[1] “Normal Paths”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/normal-paths.pdf

[2] “Surjective function”
Wikipedia
https://en.wikipedia.org/wiki/Surjective_function

[3] “Domain of a function”
Wikipedia
https://en.wikipedia.org/wiki/Domain_of_a_function

[4] “Partial function”
Wikipedia
https://en.wikipedia.org/wiki/Partial_function

[5] “Function application”
Wikipedia
https://en.wikipedia.org/wiki/Function_application

[6] “Currying”
Wikipedia
https://en.wikipedia.org/wiki/Currying