Natural Univalent Calculus

by Sven Nilsen, 2022

In this paper I introduce a calculus of closed natural numbers where equality equals univalence.

The Natural Univalent Calculus is given by the following functions and an initial object `0`:

$$\begin{aligned} &\text{add} : \mathbb{N}_c \times \mathbb{N}_c \to \mathbb{N}_c && \text{mul} : \mathbb{N}_c \times \mathbb{N}_c \to \mathbb{N}_c \\ &\text{even} : \mathbb{N}_c \to \mathbb{N}_c && \text{comp} : \mathbb{N}_c \to \mathbb{N}_c \\ & 0 : \mathbb{N}_c \end{aligned}$$

The symbol `$\mathbb{N}_c$` means closed natural numbers[1].
The function `even` returns `1` if the input is even and `0` otherwise.
The function `comp` inverts every bit in the binary form of the closed natural number.
The only way to construct `1` is by `even(0)`, since `add(0, 0) = 0, mul(0, 0) = 0, comp(0) = 0`.
This means the only way to create involutions[2] is through `even`.

Natural Univalent Calculus is special because it is impossible to construct total equality.
This is because the only way to "read" the information back from computations is through `even`.
For `add` and `mul`, the output is always greater or equal compared to any input. This means `even` controls both folding backs and involution in the calculus.

To tell that `x == x`, requires knowing that each bit of one argument is equal to the corresponding bit in the other argument. This can only be proved by applying `even` over and over using some algorithm. Thus, the algorithm that checks for equality must halt in order to make sense.

As a consequence, `x == x` can not terminate in general, only for finitely constructed natural numbers. This means that there must exist some expression in Natural Univalent Calculus that expresses the number. However, `x != y` will halt eventually.

One can imagine having higher cardinalities of time[3]. The same relation holds: `x == x` will never terminate, even when including hypertasks, but `x != y` will halt eventually.

This halting property of Natural Univalent Calculus means that one can tell `x == x` by using a meta-algorithm deciding whether the program will halt or not. Since the halting problem is undecidable[4], it is not possible to create a universal algorithm that computes `x == x`.

Symbolic Distinction[5] is an partial binary operator that halts when the inputs are symbolic distinct, but does not halt otherwise. In the Path Semantical model of Symbolic Distinction in IPL[6], one can prove that Symbolic Distinction equals Univalence[7]:

∴	eqq == univ	
∴	eq_eq_q == univ	by `eqq <=> eq_eq_q`
∴	univ == univ	by `eq_eq_q <=> univ`
∴	true	

∵	eqq <=> eq_eq_q	Symbolic distinction, equals, equality equal to quality
∵	eq_eq_q <=> univ	Equality equal to quality, equals, univalence

Therefore, equality in Natural Univalent Calculus equals univalence.

Some useful functions and objects:

```
1 := even(0)
inc := \(x) = add(x, 1)
not := even
id := \(x) = x
eqb := even . add
and := mul
or := \(x, y) = not(and(not(x), not(y)))
imply := \(x, y) = or(not(x), y)
nand := even . mul
if := \(x, y) = \(c) = add(mul(c, x), mul(not(c), y))
sub := \(x, y) = add(x, comp(y))
is_non_zero := \(x) = if(even(add(comp(x), 1)), 1)(even(x))
is_zero := not . is_non_zero
eq := \(x, y) = is_zero(sub(x, y))

is_ge_two := \(x) = and(even(x), is_non_zero(x))
is_lt_two := not . is_ge_two
is_one := \(x) = not(or(even(x), is_ge_two(x))
is_ge_three := \(x) = and(not(even(x)), is_ge_two(x))
is_lt_three := not . is_ge_three
is_two := \(x) = not(or(is_lt_two, is_ge_three))
is_ge_four := \(x) = and(even(x), is_ge_three(x))
is_lt_four := not . is_ge_four
is_three := \(x) = not(or(is_lt_three, is_ge_four))
is_ge_five := \(x) = and(not(even(x)), is_ge_four(x))
is_lt_five := not . is_ge_five
is_four := \(x) = not(or(is_lt_four, is_ge_five))
sub4 := \(x) = if(if(3, 0)(is_two(x)), if(2, 1)(is_one(x)))(is_ge_two(x))
eq4 := \(x, y) = or(
        or(eqb(is_zero(x), is_zero(y)), eqb(is_one(x), is_one(y))),
        or(eqb(is_two(x), is_two(y)), eqb(is_three(x), is_three(y)))
)
```

For any finite set of closed natural numbers, one can define substraction and equality. However, for infinite sets of natural numbers, this is not possible, because it can not terminate for all inputs. In particular, it is impossible to prove `x == x` for all natural numbers `x`.

The `comp` function has the same halting property as `x == x`:

- If `x == x` halts, then `comp(x)` halts
- If `x == x` does not halt, then `comp(x)` does not halt

It is impossible to prove `x == x` without using the `comp` function.

Some useful normal paths[8]:

```
add[even] => eqb                    and[not] => or
mul[even] => or                     or[not] => and

comp[(even, is_zero) → even] => eqb
```

References:

[1] "Closed Natural Numbers"
Sven Nilsen, 2020
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/closed-natural-numbers.pdf

[2] "Involution (mathematics)"
Wikipedia
https://en.wikipedia.org/wiki/Involution_(mathematics)

[3] "Hypercomputation"
Wikipedia
https://en.wikipedia.org/wiki/Hypercomputation

[4] "Halting problem"
Wikipedia
https://en.wikipedia.org/wiki/Halting_problem

[5] "Symbolic Distinction"
Sven Nilsen, 2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/symbolic-distinction.pdf

[6] "Prop"
AdvancedResearch – Propositional logic with types in Rust
https://github.com/advancedresearch/prop

[7] "univalence axiom"
nLab
https://ncatlab.org/nlab/show/univalence+axiom

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