Dit Calculus

by Sven Nilsen

In this paper I introduce a calculus for counting different items ("dit") that supports superposition using a sophisticated combination of ideas from Exponential Propositions, Path Semantics, Homotopy Type Theory, Cubical Type Theory and Path Semantical Quantum Propositional Logic.

Assume that there is a list `x`:

```
x := [1, 2, 3]
```

The function `dit` takes a list and counts the number of different items:

```
dit : [T] → nat
```

Such that `dit(x) == 3`. Another example is `dit([1, 2, 2]) = 2`.

When one knows the number of different items in two lists `x, y`, one can predict the bounds on the number of different items in the list `concat(x, y)`[1][2]:

```
        x : [dit] a                y : [dit] b
------------------------------------------------------------
concat(x, y) : [dit] ((>= max2(a, b) & (<= (a + b))
```

The notion of equality used in `dit` is simply `a == b`.
It means, that `a` and `b` are treated as equal, when `a == b`,
no matter where they are located within the list.

Now, I am going to replace `dit` by some imaginary function `dit_exp` which uses a different notion of equality that depends on the index and supports superposition[3][4][6]:

```
(x, y) : [dit_exp] <i, j> ((>= (dit_exp(x ∧ y) @ (i, j))) & (<= (dit_exp(x ∨ y) @ (i, j))))

(x ∧ y) : [dit_exp] max2(dit_exp(x), dit_exp(y))     with HOOO
(x ∨ y) : [dit_exp] (dit_exp(x) + dit_exp(y))        with HOOO
```

The notion of equality used here, is where `a` is located at index `n` and `b` is located at index `m`:

```
a^n == b^m    <=>    (a == b) & (n == m)

n : nat
m : nat
```

The `dit_exp` function is often omitted such that one can use the `@` operator directly:

```
x @ i        <=>        dit_exp(x) @ i          x : [T]
```

For example, `([1, 2, 3] @ 0) == 3` and `([1, 2, 3] @ 1) == 3`, since there is none superpositions within the list, so the lower and upper bounds are the same. This is true for all normal concrete lists.

Here is another example:

([1], [1]) @ 0) == 1
(([1], [1]) @ 1) == 2

This is because I can get `[1]` or `[1, 1]` from `([1], [1])`.
In the first case of `[1]`, there is only one item.
In the second case of `[1, 1]`, there are two items,
which might be counted as one or two, depending on how one interprets equality of indices.

Instead of having a fixed interpretation of how to count different items in a list, I develop a language to express what I mean by two items being different. This language is more flexible than normal equality where all items are counted as different if they are not equal.

To express that all `1`s are counted as one:

1 ~~ 1

This means `1` is qual to `1` (notice that the “e” is missing in “**e**qual”).

When I have the following axiom:

∀ x { x ~~ x }

I reduce `dit_exp` to the language of `dit`.

To express that all `1`s are counted differently:

1 ~¬~ 1

This means `1` is aqual to `1` (notice that the “e” is replaced by “a” in “**e**qual”).

When I have the following axiom:

∀ x { x ~¬~ x }

I reduce `dit_exp` to the language of `len`, which measures the length of a list.

When I have the following axiom:

∀ x, y { x ~~ y }

I collapse `dit_exp` to the language of `(>= 0) · len`,
which only distinguishes empty lists from non-empty lists.

When I have the following axiom:

∀ x, y { x ~¬~ y }

I get the following property:

∀ x, i { (x @ i) == (x @ 1) }

The `dit_exp` operator has an implicit argument which is an empty list of assumptions:

dit_exp{[]} <=> dit_exp   assumptions are empty, unless specified otherwise

For example, when I use the assumption `∀ x { x ~~ x }`, I can reduce to `dit`:

dit_exp{[∀ x { x ~~ x }]} <=> dit

Notice that since this calculus operates on lists and lists can be used as assumptions,
it is possible to create very sophisticated expressions where the assumptions are in superposition.

The `~~` operator and `~¬~` follows these rules in Path Semantical Quantum Propositional Logic[7]:

(a ~~ b) == ((a == b) & ~a & ~b)

(a ~¬~ b) == ((a == b) & ¬~a & ¬~b)

~a == (a ~~ a)

~a & (a == b)^true => ~b

However, the axiom `¬~a == ~¬a` is removed, since it does not make sense.

For example, if `a ~~ b`, then all `a`s are counted as one and all `b`s are counted as one, since `a`s and `b`s together are counted as one.

Another example, if `a == b` and all `a`s are counted as one and all `b`s are counted as one, then all `a`s and `b`s together are counted as one `a ~~ b`.

If all `a`s are counted as one and you can prove `a == b` without any assumptions,
then all `b`s are counted as one.

Now, you might wonder what is the meaning of all this. What is the utility of using this calculus?

The utility is that I can describe objects which can overlap with each other to some degree, perhaps in many ways, without needing to specify precisely how they overlap. I also do not have to specify the space in which these objects overlap. I can compute the bounds on this overlap, which provides me valuable information on how these objects are constrained.

x ∧ y   Maximum overlap
x ∨ y   Minimum overlap

Notice how similar this is to `and` and `or`. However, here, we can think of `x` and `y` as dynamic, since they together form a superposition. When we measure maximum overlap, we “move” the objects together, instead of treating them as static objects. Similarly, when we measure minimum overlap, we “move” the objects apart.

References:

[1] "Alphabetic List of Functions"
Standard Dictionary for Path Semantics
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/alphabetic-list-of-functions.pdf

[2] "Sub-Types as Contextual Notation"
Sven Nilsen, 2018
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/sub-types-as-contextual-notation.pdf

[3] "Hooo – Propositional logic with exponentials"
AdvancedResearch
https://github.com/advancedresearch/hooo

[4] "Higher Order Operator Overloading"
AdvancedResearch – Reading sequence on Path Semantics
https://github.com/advancedresearch/path_semantics/blob/master/sequences.md#higher-order-operator-overloading

[5] "Homotopy Type Theory"
Website for Homotopy Type Theory
https://homotopytypetheory.org/

[6] "cubical type theory"
nLab
https://ncatlab.org/nlab/show/cubical+type+theory

[7] "PSQ – Path Semantical Quantum Propositional Logic"
AdvancedResearch – Summary page on path semantical quality
https://advancedresearch.github.io/quality/summary.html#psq---path-semantical-quantum-propositional-logic