Joker Calculus

by Daniel Fischer, William Alexander Morris, Sven Nilsen, 2021-2023

In this paper we present open and closed calculus of nested Joker/Seshatic/Platonic languages.

The paper “Seshatism”[1] outlined an alternative way of thinking about existence to Platonism. Seshatism is the dual of Platonism and these two might share a dynamic relation[2].

The paper “The Joker”[3], discussed a mathematical universe where messages are not taken seriously. A problem we have been working on, is to figure out how The Joker fits in the picture of mathematical languages compared to Seshatism and Platonism.

We found a calculus by the following grammar,
called “Joker Calculus” (JC) with variants “Open” (OJC) and “Closed” (CJC):

expr ::= 0 | 1 | !expr | id(expr) | ?expr | (expr expr) | (expr, expr)

0 = Seshatic/Seshatism
1 = Platonic/Platonism
? = Joker
id = terminal evaluation of sub-expression (preserves inner expression)

In Closed Joker Calculus, higher dualities are involutions.
In Open Joker Calculus, higher dualities are adjoints.

The basic principle is that languages can have a depth level and surface level:

(<depth level>, <surface level>)

The Joker construction is a shorthand for when the surface level is the dual of the depth level:

?x <=> (x, !x)

A sequence construction `x y` provides the ability to create variants with the following property:

(x y, x z) => x (y, z)

This property assumes `x` only has one level.

The next page describes normalization rules.

Evaluation consists of 3 parts: eval_without_desequence, desequence, eval.

The following normalization rules are used by `eval_without_desequence` (CJC marked in **bold**):

0	=>	0		
1	=>	1		
id(x)	=>	id(x)		
!0’	=>	1		
!1’	=>	0		
!(!x)’	**=>**	**x**		**Closed JC**
!(!x)’	=>	!!x		
!(?x)’	=>	?eval(!x)		
!(x y)’	=>	eval((!x ?y))		
!(x, y)’	=>	eval((!x, !y))		
!id(x)’	=>	!id(x)		
?(?x)’	**=>**	**x**		**Closed JC**
?(x, ?y)’	**=>**	**eval((?x, y))**		**Closed JC**
?(?x, y)’	**=>**	**eval((x, ?y))**		**Closed JC**
?(x, y)’	=>	?(x, y)		
?x’	=>	?x		
((?x)’, (?!x)’)	=>	eval(?(x, !x))		
((0 x)’, 0’)	**=>**	**eval(0 (x, ?1))**		**Closed JC**
(0’, (0 x)’)	**=>**	**eval(0 (?1, x))**		**Closed JC**
((1 x)’, 1’)	**=>**	**eval(1 (x, ?0))**		**Closed JC**
(1, (1 x)’)	**=>**	**eval(1 (?0, x))**		**Closed JC**
((x y)’, (x z)’)	=>	eval(x (y, z))		
(x’, x’)	=>	x		
((!y)’, y’)	=>	eval(?!y)		
(x’, (!x)’)	=>	eval(?x)		
(x’, y’)	=>	(x, y)		
x’ x’	=>	x		
x' (y z)'	=>	(x y) z		
(x y)' z’	if eval(y z) == 0	=>	(x 0)	
(x y)' z'	if eval(y z) == 1	=>	(x 1)	
0’ (x y)’	=>	eval((0 x) y)		
1’ (x y)’	=>	eval((1 x) y)		
0’ (?1)’	**=>**	**0**		**Closed JC**
0’ (?0, 0)’	**=>**	**0**		**Closed JC**
0’ (0, ?0)’	**=>**	**0 1**		**Closed JC**
0’ x’	=>	0 x		
1’ (?0)’	**=>**	**1**		**Closed JC**
1’ (?1, 1)’	**=>**	**1**		**Closed JC**
1’ (1, ?1)’	**=>**	**1 0**		**Closed JC**
1’ x’	=>	1 x		
x’ y’	=>	x y		

Matching goes from top to bottom, `x’` means that `x` has been evaluated before matching.

The `desequence` algorithm removes repeating patterns in sequences:

```
pub fn desequence(&self) -> Expr {
    let mut s = vec![];
    self.sequence(&mut s);
    if s.len() == 1 {return self.clone()};

    let mut changed = false;
    let mut n = 1;
    loop {
        if 2 * n > s.len() {break}
        for i in 0..s.len() + 1 - 2 * n {
            if (0..n).all(|k| s[i+k] == s[i+k+n]) {
                for k in (0..n).rev() {
                    s.remove(i+k);
                }
                changed = true;
                break;
            }
        }
        if changed {
            changed = false;
            n = 1;
        } else {
            n += 1;
        }
    }

    s.reverse();
    let mut exp = s.pop().unwrap();
    while let Some(x) = s.pop() {
        exp = seq(exp, x);
    }

    exp
}
```

Where `sequence` flattens a sequence:

```
pub fn sequence(&self, s: &mut Vec<Expr>) {
    match self {
        Seq(a, b) => {
            a.sequence(s);
            b.sequence(s);
        }
        _ => s.push(self.clone()),
    }
}
```

Finally, evaluation is done by `eval`:

```
pub fn eval(&self, closed: bool) -> Expr {
    let mut v = self.eval_without_desequence(closed).desequence();
    loop {
        let w = v.eval_without_desequence(closed).desequence();
        if w == v {return v}
        v = w;
    }
}
```

Any two expressions that normalize to one same expression are equal.

Joker Calculus is associative with respect to sequences:

$$x\ y\ z = (x\ y)\ z = x\ (y\ z)$$

With Closed Joker Calculus one can prove that `!!x == x` for all `x`.
This makes Closed Joker Calculus a generalization of classical propositional logic.
However, this does not hold in Open Joker Calculus, which might be thought of as constructive.

The Joker Calculus can be used to analyze language of some jokes.
For example, irony[4] might be thought of as “Joker Seshatism”:

?1	<=>	Joker Seshatism	<=>	Irony
?1 = (1, 0)	<=>	Joker Seshatism = Irony = (Seshatism, Platonism)		

The first component in a tuple is called a “depth” language.
The second component in a tuple is called a “surface” language.

Irony can be thought of as Joker Seshatism because at the surface, the language pretends to say something that might be interpreted to be literally true (Platonism), while the depth of the language contradicts the surface language (Seshatism).

A less known form of humour is Joker Platonism, which at surface level pretends to say something that is different by comparison (Seshatism), while the depth of the language contradicts the surface language (Platonism) by “cancelling” the difference.

For example, a joke using Joker Platonism:

> my natural numbers are bigger than yours

Natural numbers can be mapped onto themselves in an almost one-to-one correspondence such that every number on one side is greater than at least one number on the other side. However, this requires at least one number on the other side does not have a partner number on the greater side.

This mean, although one person can have natural numbers that are “bigger” than the natural numbers of another person, it does not matter because all such number systems are equivalent.

Joker Calculus also supports combinations such as:

Seshatic Platonism	Platonism viewed from a perspective of Seshatism
Platonic Seshatism	Seshatism viewed from a perspective of Platonism

The normalization rules ensures that every redundant sense gets reduced, such as:

Seshatic Seshatism	=>	Seshatism
Platonic Platonism	=>	Platonism
(Seshatism, Seshatism)	=>	Seshatism
(Platonism, Platonism)	=>	Platonism

One can use Joker Calculus to find new levels of language, such as:

> (Seshatic Platonism, Platonic Seshatism)

This language does not reduce further, so it is a different language than e.g. Joker Platonism.
At the surface level, this language is Platonic Seshatism, but in depth it is Seshatic Platonism.

For example, a physicist might use an informal mathematical language with rigor along some dimensions but not others, having a kind of Platonic Seshatism. However, the theory that the physicist is working against, might be viewed as more rigorous, yet with some dynamic elements, which is a kind of Seshatic Platonism.

References:

[1] “Seshatism”
Sven Nilsen, 2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/seshatism.pdf

[2] “Seshatic-Platonic Cycles”
Sven Nilsen, 2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/seshatic-platonic-cycles.pdf

[3] “The Joker”
Sven Nilsen, 2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/the-joker.pdf

[4] “Irony”
Wikipedia
https://en.wikipedia.org/wiki/Irony