HOOO Exponential Propositions

by Sven Nilsen, 2022

In this paper I present axioms of Exponential Propositions using the semantics of Higher Order Operator Overloading. I also explain the motivation and development of these axioms.

Axioms for HOOO Exponential Propositions where `□` means binary operator `□[¬]` dual operator:

```
a^b => (a^b)^c
(a □ b)^c == (a^c □ b^c)
c^(a □ b) == (c^a □[¬] c^b)
```

For example, `and[not] <=> or`, therefore `or` is the dual operator of `and`.

The axioms that use `□` or `□[¬]` are called “axiom schemes”.

These axioms are not completely figured out: It is currently unknown which axioms are constructive and which are classical (requries excluded middle). Since this requires further work, I decided to continue to work on this as part of the development of the Prop[1] library and update this paper when finished. At this moment, I am guessing that the non-dual axiom scheme does not require excluded middle.

For example, the following axiom is believed to be classical:

```
c^(a ∧ b) => (c^a ∨ c^b)      (a ∨ ¬a)  (b ∨ ¬b)  (c ∨ ¬c)
```

The reverse has a constructive proof:

```
∴     (c^a ∨ c^b) => c^(a ∧ b)

∵     (a^c ∧ b^c) => (a ∧ b)^c
∵     (a => b)^c => (a^c => b^c)
```

This proof holds if and only if the two axioms below holds.

It is not necessary to start with the full two axiom schemes, because some axioms within the schemes can be derived from the others. However, these axioms schemes are considered the axioms of HOOO Exponential Propositions due to the symmetry.

It might be surprising to somebody that symmetry and not common sense intuition should guide the choice of axioms for some theory. I am simply going to disregard that view and go for symmetry. I like the axioms this way because they are operational and minimalistic in some sense. I can prove a lot of theorems, even derive axioms for Modal Logic[2].

The `^` operator can be thought of as a reverse turnstile `⊢` binary operator, or the sequent[3]:

```
⊢ (x => y)     <=>  y^x
```

The syntactical choice of `^` is ergonomic as it has an intuitive operator presedence in algebra.

Translated, the axioms can be represented using sequents:

```
⊢ (⊢ (b => a)) ∧ (d ⊢ c) => (⊢ (c => ⊢ (b => a)))
⊢ (c ⊢ (a □ b)) == ((c ⊢ a) □ (c ⊢ b))
⊢ (a □ b) ⊢ c) == ((a ⊢ c) □[¬] (b ⊢ c))
```

The rest of this paper explain what the `^` operator means, why and how it was developed.

Function pointers as exponential objects

Higher Order Operator Overloading[4] (HOOO) is a technique where operators (usually binary) are extended to functions that operate on the same source of data. This technique is often used to simplify proofs both conceptually and syntactically[5]. More about this will come later in the paper.

It turns out that in proof semantics, there is a significant distinction between lambdas (or closures) and function pointers that are not allowed to capture variables from the environment. Assume that there is a function pointer `f` of type `A → B` and a lambda `g` of type `A → B`:

`f : fn(A) → B`	`f` is a function that does not capture from the environment
`g : \(A) → B`	`g` is a lambda that can capture from the environment

In general, when this distinction is irrelevant, one simply talks about functions:

`h : A → B`	`h` might be a function pointer or lambda

Using the Curry-Howard correspondence[6], `f, g, h` are proofs of `A => B` in constructive logic[7]. Yet, the proof strength varies between `f` and `g, h`.

For every function pointer, there exists a corresponding lambda, but not vice versa. Therefore, `f` is a proof of a stronger statement of `A => B` as a proposition compared to `g` and `h`. In general, one can use lambdas only when the distinction between function pointers and lambdas is irrelevant.

Now, using exponential objects[8] from Category Theory, I introduce a new operator `^`:

`B^A`	`B` is provable from `A` without any further assumptions

Notice that when using `→` one writes `A` first and `B` second, while the `^` operator writes `B` first and `A` second. When working with `^` only, in logical notation without proofs, it is common to write e.g. `a^b` when `a` is provable from `b`, to make it easier to read (like ordinary algebra).

Another intuition of `^` is that one can think about it as a "where" statement.

For example, in standard mathematics it is common to make statements such as:

1 / x   where x != 0

The only difference is that "where" is replaced by `^` and both sides need to be propositions.

Function pointers as propositions

The idea is to start thinking about \`a^b\` as a proposition itself. I realized that this could be used to solve a problem in Path Semantics[9], where the qubit operator \`~\`[10] requires a stronger notion of equality to allow substitution in the argument:

```
~a ∧ (a == b)^⊤  =>  ~b
```

Here, \`(a == b)^⊤\` means that \`a == b\` can be proved without needing any assumptions. The symbol \`⊤\` means "true" and it works because nothing can be proved from "true", which is not already provable without making any assumptions.

This stronger notion of equality corresponds to tautological equality.

In classical logic, one can think about \`~a\` as generating a new pseudo-random[11] proposition using \`a\` as a seed. This works by depending on the entire bit sequence of \`a\`, but in a such way that the order of bits is not revealed. This is possible because in normal logic there are no branches, so in principle one can check the entire proof in \`O(1)\` using bit sequences for each argument proposition. A little counter-intuitively, even \`~\` takes only a single bit at a time, one can not substitute \`a\` with \`b\` using \`a == b\`. It is the randomness that makes the \`~\` operator depending on the entire bit sequence of \`a\`. However, if \`b\` is provably equal to \`a\` without any assumptions, then this operation is allowed.

Logicians might recognize the statements above as sequents[3], or natural deductions[12]:

HOOO	**Sequent**	**Natural deduction**	**Name**
`x^⊤`	`⊢ x`	`⊢ x`	`tautology`
`⊥^x`	`x ⊢`	`x ⊢ ⊥`	`paradox`

One difference is that \`^\` is a binary propositional operator. A binary propositional operator requires two arguments, which both need to be propositions.

A sequent can have multiple arguments at both sides, or even zero arguments, which might not be propositions. A natural deduction can have multiple arguments on the left side, or even zero arguments, but needs an argument on the right side.

Another difference is that the semantics of \`^\` is grounded in function pointers. Sequents and natural deductions are grounded in proof semantics. The one-to-one translation between function pointers and proofs is what the Curry-Howard correspondence is about.

Yet, it is possible that the choice of axioms might yield interesting theories using function pointers, that have no direct analogue in proof semantics. This is about leveraging function pointers as exponential objects in the way they differ from lambdas.

When a programming language using a distinction between function pointers and lambdas is formalized in logic, the formation rules of \`^\` follows from which function pointers can be constructed. This corresponds to which programs are valid in a given programming language.

However, since function pointers can be passed around by arbitray complex algorithms, it is non-trivial which programs correspond to valid proofs. The type system in many programming languages do not check all properties of programs required for the Curry-Howard correspondence to hold in all cases.

Checking proofs for circular reasoning (recursion)

When I developed the axioms for HOOO Exponential Propositions, I had to write a script (using Dyon[13]) that checks the code for the HOOO module that no theorem calls itself recursively. The library being developed is Prop[1] (Propositional Logic with types in Rust).

For example, when a function calls itself, it can be used to prove `b^a` for any `b`:

```
    f(x : A) := f(x)         f : A → B
```

This is not a valid proof, but in programming languages where self recursion is allowed, there might be no check that `f` will terminate. In fact, this problem is unsolvable (undecidable) in general, known as the Halting problem[14].

Theorem proving using HOOO Exponential Propositions

The insight that lead to the development of HOOO Exponential Propositions, was the idea that the propositional semantics of `^` could use the semantics of Higher Order Operator Overloading[4].

For example:

```
    a^c == b^c
```

Here, I state that the propositional truth of `a^c` equals the propositional truth of `b^c`. This mode of reasoning is difficult to wrap one's head around in the beginning because one is used to think about propositions like `a == b`. To understand what is going on here, I will translate the above to lambdas:

```
∴   f == g  <=>  \(x) = (f(x) == g(x))

∵   f : fn(c) → a      g : fn(c) → b
```

This is a commonly used technique in Path Semantics, which is HOOO.

Now, instead of HOOO generating lambdas only, I refine the overloading on function pointers:

```
∴   (f == g)  <=>  fn(x) = (f(x) == g(x))

∵   f : fn(c) → a      g : fn(c) → b
```

It works, because for every function pointer there is a lambda. The type refinement is backwards compatible with old use of HOOO.

However, there is a problem: `f(x) == g(x)` is not constructive. In normal programming languages this would return a `bool`. What I want is a type level proposition. This is fixed by lifting the new refined overloading to types:

```
    (a^c == b^c) == (a == b)^c
```

The expression `a^c == b^c` means that if I have `a^c` I can produce `b^c` and vice versa.
If this is possible, then I can also produce an expression `(a == b)^c`.
This proves `(a^c == b^c) => (a == b)^c`.

Likewise, if I have `(a == b)^c` and ` a^c` I can produce `b^c`.
If I have `(a == b)^c` and `b^c` I can produce `a^c`.
This proves `(a == b)^c => (a^c == b^c)`.

Now, it turns out that one can use the following:

```
        (a^c => b^c) == (a => b)^c
```

and:

```
        a^b => (a^b)^c
```

To prove:

```
        (a^c == b^c) == (a == b)^c
```

However, it is far from trivial how to do this. Let me show you how complex it is.

In Prop[1], the source code for this proof is the following:

```
        /// `(a == b)^c => (a^c == b^c)`.
        pub fn hooo_eq<A: Prop, B: Prop, C: Prop>(
                x: Pow<Eq<A, B>, C>
        ) -> Eq<Pow<A, C>, Pow<B, C>> {
                fn f<A: Prop, B: Prop>(x: Eq<A, B>) -> Imply<A, B> {x.0}
                fn g<A: Prop, B: Prop>(x: Eq<A, B>) -> Imply<B, A> {x.1}
                let x1 = pow_transitivity(x.clone(), f);
                let x2 = pow_transitivity(x, g);
                (hooo_imply(x1), hooo_imply(x2))
        }

        /// `(a^c == b^c) => (a == b)^c`.
        pub fn hooo_rev_eq<A: Prop, B: Prop, C: Prop>(
                x: Eq<Pow<A, C>, Pow<B, C>>
        ) -> Pow<Eq<A, B>, C> {
                let x1 = hooo_rev_imply(x.0);
                let x2 = hooo_rev_imply(x.1);
                hooo_rev_and((x1, x2))
        }

        /// `b^a ∧ c^b => c^a`.
        pub fn pow_transitivity<A: Prop, B: Prop, C: Prop>(
                ab: Pow<B, A>,
                bc: Pow<C, B>,
        ) -> Pow<C, A> {
                fn f<A: Prop, B: Prop, C: Prop>(
                        a: A
                ) -> Imply<And<Pow<B, A>, Pow<C, B>>, C> {
                        Rc::new(move |(ab, bc)| bc(ab(a.clone())))
                }
                hooo_imply(f)(hooo_rev_and((pow_lift(ab), pow_lift(bc))))
        }

        /// `(a^c ∧ b^c) => (a ∧ b)^c`.
        pub fn hooo_rev_and<A: Prop, B: Prop, C: Prop>(
                x: And<Pow<A, C>, Pow<B, C>>
        ) -> Pow<And<A, B>, C> {
                let g = pow_to_imply(hooo_imply);
                let f = pow_to_imply_lift(imply::and_map);
                let f = imply::transitivity(hooo_imply(f), g);
                f(x.0)(x.1)
        }

        /// `b^a => (a => b)`.
        pub fn pow_to_imply<A: Prop, B: Prop>(pow_ba: Pow<B, A>) -> Imply<A, B> {
                Rc::new(move |a| pow_ba(a))
        }
```

```
/// `b^a  =>  (a => b)^c`.
pub fn pow_to_imply_lift<A: Prop, B: Prop, C: Prop>(
        pow_ba: Pow<B, A>
) -> Pow<Imply<A, B>, C> {
        fn f<A: Prop, B: Prop, C: Prop>(
                _: C
        ) -> Imply<Tauto<Imply<A, B>>, Imply<A, B>> {
                Rc::new(move |x| x(True))
        }
        hooo_imply(f)(pow_lift(pow_to_tauto_imply(pow_ba)))
}

/// `b^a => (a => b)^true`.
pub fn pow_to_tauto_imply<A: Prop, B: Prop>(
        x: Pow<B, A>
) -> Tauto<Imply<A, B>> {
        fn f<A: Prop, B: Prop>(_: True) -> Imply<Pow<B, A>, Imply<A, B>> {
                Rc::new(|pow_ba| Rc::new(move |a| pow_ba(a)))
        }
        let f: Imply<Pow<Pow<B, A>, True>, Tauto<Imply<A, B>>> = hooo_imply(f);
        f(pow_lift(x))
}
```

The axioms:

```
/// `a^b => (a^b)^c`.
pub fn pow_lift<A: Prop, B: Prop, C: Prop>(
        _: Pow<A, B>
) -> Pow<Pow<A, B>, C> {
        unimplemented!()
}

/// `(a => b)^c => (a^c => b^c)`.
pub fn hooo_imply<A: Prop, B: Prop, C: Prop>(
        x: Pow<Imply<A, B>, C>
) -> Imply<Pow<A, C>, Pow<B, C>> {
        unimplemented!()
}

/// `(a^c => b^c) => (a => b)^c`.
pub fn hooo_rev_imply<A: Prop, B: Prop, C: Prop>(
        x: Imply<Pow<A, C>, Pow<B, C>>
) -> Pow<Imply<A, B>, C> {
        unimplemented!()
}
```

The expression `unimplemented!()` makes the code pass the type checker in Rust.
This is the analogue of introducing an axiom.

Summary

HOOO Exponential Propositions allows …:

- … distinguishing function pointers from lambda expression
- … correct substitution in the path semantical qubit `~` operator
- … non-trivial theorem proving

Future work

Provability proofs similar to HOOO Exponential Propositions can also be achieved by using a Modal Logic.

Currently, the exact relationship between HOOO Exponential Propositions and Modal Logic is unknown.

For example, which version of Modal Logic? Is there a Modal Logic that covers HOOO EP?

References:

[1] “Prop”
AdvancedResearch – Propositional logic with types in Rust
https://github.com/advancedresearch/prop

[2] “Modal logic”
Wikipedia
https://en.wikipedia.org/wiki/Modal_logic

[3] “Sequent”
Wikipedia
https://en.wikipedia.org/wiki/Sequent

Paper: G. Gentzen, "Untersuchungen über das logische Schliessen" *Math. Z.* , **39** (1935)
English translation: The collected papers of Gerhard Gentzen,
North-Holland, 1969; edited by M.E. Szabo

[4] “Higher Order Operator Overloading”
Sven Nilsen, 2018
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/higher-order-operator-overloading.pdf

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

[6] “Curry-Howard correspondence”
Wikipedia
https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence

[7] “Intuitionistic logic”
Wikipedia
https://en.wikipedia.org/wiki/Intuitionistic_logic

[8] “exponential object”
nLab
https://ncatlab.org/nlab/show/exponential%20object

[9] “Path Semantics”
Sven Nilsen, 2016-2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/path-semantics.pdf

[10] “Path Semantical Qubit”
Sven Nilsen, 2022
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/path-semantical-qubit.pdf

[11] “Pseudorandomness”
Wikipedia
https://en.wikipedia.org/wiki/Pseudorandomness

[12] “Natural deduction”
Wikipedia
https://en.wikipedia.org/wiki/Natural_deduction

Paper: G. Gentzen, "Untersuchungen über das logische Schliessen" *Math. Z.* , **39** (1934)

[13] “Dyon”
PistonDevelopers
https://github.com/pistondevelopers/dyon

[14] “Halting problem”
Wikipedia
https://en.wikipedia.org/wiki/Halting_problem