HOOO Exponential Propositions

by Sven Nilsen, 2022-2023

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 Higher Order Operator Overloading Exponential Propositions (HOOO EP):

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

The symbol `⊤` stands for “true” (unit type) and `⊥` for “false” (empty type).

HOOO EP makes it possible to do theorem proving in the style of Sequent Calculus[3],
but by using propositions instead of sequents.

The expression `a^b` means a function pointer (not lambda expression) of type `b → a`.
This distinction from lambda expression is important, since function pointers can not capture from the environment. HOOO EP provides inference rules to existing type systems that distinguishes between function pointers and lambda expressions, which can reason about its own theorems.

Basically, HOOO EP makes it possible to reason about which function pointers can be constructed within a given programming language, like Rust, using its own type system, by adding a few unimplemented functions (using source from the Prop[1] library):

```
/// `a^b`.
pub type Pow<A, B> = fn(B) -> A;

/// A tautological proposition.
pub type Tauto<A> = fn(True) -> A;

/// `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)^true`.
pub fn tauto_hooo_imply<A: Prop, B: Prop, C: Prop>(
	_: Pow<Imply<A, B>, C>
) -> Tauto<Imply<Pow<A, C>, Pow<B, C>>> {
	unimplemented!()
}

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

The `unimplemented!()` expression is like introducing an axiom. The rest of the “hooo” module in Prop is a complex collection of theorems built on top of these axioms.

The concrete programming understanding of HOOO EP is relatively easy: “pow_lift” says that one can return a function pointer from functions. “tauto_hooo_imply” says that if one can create a lambda `a => b` from `c`, then one can create a lambda `a^c => b^c` from no arguments (`true`). “tauto_hooo_or” says that one needs values for enum variants (since `or` is an enum).

However, philosophically, HOOO EP is very hard to understand.
This is because there is a lot to take in and can easily get overwhelming.

In order to understand it philosophically,
is better to retrace the development by starting with a simple question:

    a == b

What does this mean?

In Propositional Logic, using lambda calculus equipped with types, `a == b` are two lambdas, one `a => b` and one `b => a`. Since lambdas can capture from the environment, this does not tell us anything about the actual truth of `a == b`, because if the environment contains `false`, then it can prove anything.

This problem, that `a == b` is not knowable in normal Propositional Logic, requires extending it with a new operator `^` such that one can express a stronger statement of equality `(a == b)^⊤`. This means, that `a == b` can be constructed from no arguments (`⊤` means "true" that is also like an empty environment).

In the empty environment, one can only use static functions to create lambdas of particular types. This is why function pointers are important, because they say, or assume, that something is possible without making any further assumptions.

Instead of over-thinking this, let me provide an example: When we say `a == b` in natural language, we are often saying `(a == b)^⊤` implicitly, like `1 + 1 = 2` is actually `(1 + 1 = 2)^⊤`. On the other hand, it is harder to understand precisely what `a == b` means, because its truth depends on which context one refers to. In order to avoid confusing truth and "true", one uses "tautology" for the stronger notion of "true", e.g. `a^⊤` means `a` is tautological true. Correspondingly, `⊥^a` means that `a` is paradoxically false. Paradoxically false is a stronger notion of "false" which means that it can not be true in any situation (logicians like to say "possible world").

Now, the reason equality `a == b` is important, is because it means `a` can be substituted for `b` in operators that are predicates (predicates might be thought of as functions returning "true" or "false"). By encoding functions as predicates, it means one can replace arguments that are equal. However, since `a == b` means a type `a` equals a type `b`, what one is actually saying is that one can construct *new functions* from existing ones that behave like old ones but for new types. There is no actual substitution, because in order to substitute in the same function, the two equal arguments must be of the same type.

Anyway, the gist of it is that there is a weaker notion of equality `a == b` and a strong notion of equality `(a == b)^⊤`. You might ask: What is the point? Why do we need `(a == b)^⊤`?

The reason we need `(a == b)^⊤` is because there exists operators that are not predicates/functions. You know them as data structures. A data structure only preserves information when there is a way to translate back and forth between two representations of the same information. However, in order to translate back and forth, one needs algorithms. Algorithms are constructed out of function pointers, but they can be arbitrary complex due to lambdas. So, to reason about transformations of data structures, one needs a theory that can reason about which algorithms are possible to construct.

Well, this is not the entire story. Let us say we have a function that takes a number and outputs a random number using the input number as a seed. This is how qubit `~` works in Pocket-Prover[15].

When modelling qubit `~` in Prop, the value of `u64` (since Pocket-Prover packs bits in chunks of 64 bits) is lifted into a generic argument type. There is a different generic argument type for each proposition in a proof.

The Rust code for qubit `~` in Prop is the following:

```
/// Represents a recursive qubit proposition.
#[derive(Clone)]
pub struct Qubit<N, A>(N, A);
```

Notice that this is a data structure, which requires `(a == b)^⊤`.

The argument `N` is used to tell how many times the qubit operator is applied, which makes theorem proving easier. For simplicity, one could remove the argument `N`:

```
/// Represents a qubit proposition.
#[derive(Clone)]
pub struct Qubit<A>(A);
```

Now, since `A` is not `pub A`, it is not possible to create `Qubit<B>` from `Qubit<A>` outside the module “qubit” in Prop. In fact, had it not been for `qubit::in_arg`,
which restricts to `(a == b)^⊤` in order to model qubit `~` properly, there would be no way to do it.

In logic, one often assumes that everything is open and known. This is called a “perfect information environment”. However, in reality, there are boundaries of knowledge. In programming, this is useful to reduce complexity and bugs. Why is this not common in logic?

The reason is that logic developed historically as a mean to prove theorems in mathematics. For such a use case, one could not prove more theorems with boundaries of knowledge. Another reason was that mathematics is heavily invested in First Order Logic, which uses predicates only.

HOOO EP makes it possible to not only reason about function pointers globally, but also reason counter-factually (“what if” scenarios) of which algorithms would be possible, given some types of function pointers existed. This helps with reasoning about boundaries of knowledge.

The name Higher Order Operator Overloading (HOOO) comes from an idea of two axiom schemes:

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

Here, `□` means a classical binary operator (and, or, imply etc.) and `□[¬]` means the dual operator. For example, `and[not] <=> or`, therefore `or` is the dual operator of `and`.

This picture comes from an idea that operators can be lifted up and down to allow higher level reasoning. While the two axiom schemes was a good idea, it turns out that some of the axioms produces an inconsistent theory while others need to be restricted to decidable propositions (classical propositions with excluded middle). Furthermore, the right sides missed an `^⊤`.

After working out the details in late 2022, which was difficult, I ended up with the first 3 axioms. This work was done as part of the development of the Prop[1] library.

The axiom schemes were developed from a sense of symmetry instead of common sense intuition. This also provided lots of theorems that were tested before arriving at a solid foundation. These theorems also helped deriving axioms for a Modal Logic[2]. It is remarkable how far `^` goes.

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)))
⊢ (c ⊢ (a ∨ b)) => (⊢ ((c ⊢ a) ∨ (c ⊢ b)))
```

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.
The only difference is that “where” is replaced by `^` and both sides need to be propositions.
For example, in standard mathematics it is common to make statements such as:

```
1 / x    where x != 0                (1 / x)^(x != 0)    (analogue)
```

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. The idea is to "translate" from HOOO to HOOO EP:

```
(a^c == b^c) == (a == b)^c      classical proof in HOOO EP
```

Only the following direction can be proved constructively, showing that HOOO EP is non-trivial:

```
(a == b)^c => (a^c == b^c)      constructive proof in 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

[15] “Pocket-Prover”
AdvancedResearch
https://github.com/advancedresearch/pocket_prover