Single Variable First-Order Proof Transform Into Propositional Logic

by Sven Nilsen, 2020

In this paper I represent a technique to transform first-order proofs of a single variable into propositional logic, based on a kind of reverse Higher Order Operator Overloading semantics.

Henri Mauer tried to understand a trick I used in an example for the “pocket_prover” library:
https://github.com/advancedresearch/pocket_prover/blob/master/examples/exam.rs

The trick is to introduce an extra proposition `x` and use the following translation:

First-order logic:	**Propositional logic:**
∀ x { p(x) }	p
∃ x { p(x) }	x => p

On the left side, `p` is a predicate, but on the right side, `p` is a proposition.
However, this trick can only be used when the proofs uses a single variable.

I could not find where I got this trick from, but believe that I must have made it up for the example.
This paper is an attempt to explain how it works.

In first-order logic, it might seem at first that the major feature is able to express quantification over non-logical variables. However, what the semantics of first-logic does is tautologies over all possible predicates. A predicate `f` is a function of the type:

$$f : T \to \text{bool}$$

Such that a proof in first-order logic of `N` predicates and `M` propositions is of the type:

$$\text{proof} : (T \to \text{bool})^N \times \text{bool}^M \to \text{bool}$$

With the existential path equation:

$$\text{proof} <=> \backslash\text{true}$$

Hence, when `T` is a finite type, a first-order logic proof is isomorphic to a proof in propositional logic.

A variable is a map from an environment `T` to an indexed type `$X_i$`:

$$\text{variable}_i : T \to X_i$$

So, when a predicate function `p` produces a proposition by being applied to some variable `$x_i$`:

$$p(x_i) : bool$$

It is actually a function composition where the variable is produced from the environment `t`:

$$f(t) = (p \cdot variable_i)(t) = p(variable_i(t)) = p(x_i)$$

$$t : T$$

Arbitrary complex environments are constructed implicitly through quantification over variables. This construction is not finite, therefore it is not possible to enumerate all proofs in first-order logic. However, assume that one is only interested in proofs of a single variable.

For example:

$$\forall\ x\ \{\ p(x)\ \}$$

$$\exists\ x\ \{\ p(x)\ \}$$

It is not permitted to introduce more than one variable, like this:

$$\forall\ x\ \{\ \forall\ y\ \{\ p(x) \vee p(y)\ \}\ \}$$

This means that the environment `T` from which the variable is produced from is equivalent to `X`:

$$T <\sim=> X$$

Therefore, the simplest function that produces the variable from the environment is `$id_X$`:

$$f(t) = (p \cdot id_X)(t) = p(id_X(t)) = p(x)$$

$$t <=> x$$

Every expression that uses a predicate must use a single variable.

In Higher Order Operator Overloading under Boolean Algebra, operator `op` uses a single argument:

$$op(f, g) <=> \backslash(x : X) = op(f(x), g(x))$$

$$op(f, g) : X \to bool$$

$$op : X \times X \to bool$$

The insight of the transformation of proofs from first-order logic to propositional logic, when the proofs uses a single argument, comes from a reverse semantics of Higher Order Operator Overloading.

From the Curry-Howard isomorphism, a program is a proof of its type.
If any type is inhabited, it has a program, which means there exists at least one proof.

Theorem prover assistants with dependent type systems are possible,
by interpreting types are propositions where all members of the type are treated as equal:

$type_0$	<=>	prop	Types in the bottom hierarchy are propositions
x : nat nat : $type_0$			`x` is the assertion that there is a proof of `nat`
x : nat	vs	y : nat	`x` and `y` are equal in the sense they are both proofs

In programming, one does not think of `x` and `y` as equal, but for proof theory this holds.

Now, the trick is to think about the one-to-many relationship that is between types and members.
The same relationship is between operators and higher order operators introduced by overloading!

One can think of operators as types for their higher order operators.
If a program `op(a, b)` is inhabited, then the higher order program `\(x) = op(f(x), g(x))` is inhabited.
Therefore, proofs in first-order logic of a single variable are proofs of proofs in propositional logic.

Any member of a type is as good proof as any other member of the same type.
However, for higher order operators, this mean any variable is as good proof as any other variable.
Yet, if a function treats it input the same as any other input, it means the input can be erased:

p[unit → id] : () → bool

p : X → bool
unit : X → ()

A function of type `() → bool` is just `bool`.

If all members of `X` are treated as equal, then a predicate `p` has a path `p[unit → id]`.
The path `p[unit → id]` is the proposition that one wants to prove.

However, since first-order logic quantifies over variables, one must translate using the following trick:

First-order logic:	**Propositional logic:**
∀ x { p(x) }	p
∃ x { p(x) }	x => p

For example:

∃ x { a(x) ∧ b(x) ∧ ¬c(x) }	x => a ∧ b ∧ ¬c

For proofs of more than one variable, no such translation is possible in general.