Alpha Theorem Proving

by Sven Nilsen, 2018

In this paper I represent a way to do theorem proving in path semantics a bit similar to Kurt Goëdel's encoding of expressions in first order logic used in the proof of the incompleteness theorem, called "alpha theorem proving" due to a special `α` function. The `α` function operates on expressions following semantics of Alfred Tarski's truth hierarchy. This permits a style of theorem proving that is formal, yet favors mathematical intuition, because it does not require a full formal definition of path semantics nor complete knowledge about its internal laws. This result resolves a major milestone of the goals of path semantics, which is how to check path semantical proofs for soundness, given that the language of path semantics is unbounded across infinite cardinal hierarchies. This holds for any mathematical proof of finite amount of expressions.

Assume you have N mathematical objects. To describe the properties of these N mathematical objects, one uses path semantics (which is equivalent to using functions to describe things). Whatever the mathematical objects are and what one can say them, the same can be said using functions that replaces each mathematical object with a natural number. While there in principle could be possible to describe an uncountable infinite amount of objects with path semantics, such as real numbers, most proofs are only concerned by, at most, countable infinite objects as higher order relations.

With other words, there are a finite amount of expressions in the proof and what can be said about them, that is relevant, is usually countable infinite or finite. Therefore, to show that a such proof is sound, it is sufficient to show that there exists a function mapping from each expression to a natural number such that all constraints in the mathematical language are satisfied. This technique is called "alpha theorem proving" because the function mapping expressions to natural numbers uses the greek symbol `α`. One can also use another symbol or name as long it is not mixed up with other functions.

α : expression → nat

Since `α` takes an expression, the argument is quoted, but this is often assumed by treating it as special. Quotation in this context means reference to some mathematical theory at meta-level which, unless specified otherwise, follows the truth hierarchy developed by the logician Alfred Tarski. In path semantics, the mathematical theory is path semantics, so one uses alpha theorem proving in path semantics to talk about itself.

For example, you have two objects `x` and `y`:

x : bool	`x` is of type `bool`
y : bool	`y` is of type `bool`
α("bool") = α(bool) = 0	`bool` is quoted because it is not quantified
α("x") = α(x) = 1	`x` is quoted because it is not quantified
α("y") = α(y) = 2	`y` is quoted because it is not quantified
α("x : bool") = α(x : bool) = 3	`x : bool` is quoted because it is not quantified
α("y : bool") = α(y : bool) = 4	`y : bool` is quoted because it is not quantified

In the example above, every expression is assigned a number by the `α` function.

It does not matter which numbers one uses, as long one does not do things like `α(x) = 0` and `α(bool) = 0`, because a member of a type is not identical to its type, so the assigned numbers should be different.

One could also assign `x` and `y` the same number, written `α(x) = 1` and `α(y) = 1`, because there is no constraint that these mathematical objects are not the same.

Since `α` is a function in the mathematical theory itself, one must assign a number to the expression that assigns a number, recursively. One way is to do the following:

∀ a : expression { α(“α(a)”) = α(a) } `a` is not quoted because it is quantified

This assumes that the recursive hierarchy is not talked about in a way that makes the constraints inconsistent with the constraints that apply to mathematical objects in the proof.

The type `expression` of `a` can be inferred from the function type of `α`, so one can write like this:

∀ a { α(“α(a)”) = α(a) }

One should also assign a number to every operator on the numbers mapped to by `α`.

For example, all binary operators:

∀ a, b, f { α(“α(\”f(α(a), α(b))\”)”) = α(“f(α(a), α(b))”) }

f : nat × nat → any

There exists a more general way to do this. Under function currying and lambda abstraction, this forms a Cartesian closed category of everything that can be said about the objects as their proof-theoretical properties. In this category, constructed lambdas by function application are also morphisms:

f’ := \(x) = \(y) = \(z) = f(x, y, z) Cheating by turning all functions into category morphisms

f’(a)(b)(c) = f(a, b, c) Function application is a diagram in the category

All operators on the numbers mapped to by `α` are assigned recursively:

∀ a, f { α(“α(f(\”α(a)\”))”) = α(“f(α(a))”) }

Notice that this is a special case of the first recursive assignment where `a => “f(α(a))”` for some `a`:

∀ a { α(“α(a)”) = α(a) } => ∀ a, f { α(“α(f(\”α(a)\”))”) = α(“f(α(a))”) }

Therefore, it is sufficient to use the first law when convenient and then assume function currying and lambda abstraction.

However, this recursive assignment is merely a technicality, for the reason that follows.

There exists some algorithm that outputs every assignment in a proof$_0$ one by one, of the function `$\alpha_0$` and some algorithm that outputs every assignment in a proof$_1$ one by one, of the function `$\alpha_1$`:

```
∀ i, x : [α₀] i { yield x }        Generates all expressions mapped by `α₀`
∀ i, x : [α₁] i { yield x }        Generates all expressions mapped by `α₁`
```

If it is possible to run both these algorithms forward without discovering any constraint conflict for any moment in the future, then there exists a function `$\alpha_{01}$` mapping the assigned numbers in a such way that no constraint conflict happens with another. This means there exists some algorithm that outputs every assignment from `$\alpha_{01}$`:

```
∀ i, x : [α₀₁] i { yield x }       Generates all expressions mapped by `α₀₁`
```

This argument has brought us full circle mapping back to its own type, making it possible to possible to compose proofs. Notice that this is a one-to-one mapping with the natural numbers and therefore an infinite, yet countable set.

If a mathematical theory using alpha theorem proving is *already assumed* with respect to a new proof, there is no need to specify the old mapping in the context of the new proof, assuming that no such constraint conflict occurs. All constraints of the old `α` mappings carries over to the new proof.

Instead of specifying recursive constraints and language semantics for every proof, they only need to be specified once and then assumed in new proofs.

This means the previous example can be simplified to:

```
x : bool
y : bool
```

Which is identical to path semantics itself! Alpha theorem proving in path semantics means that the language of path semantics can be specified in a such way that it turns into invisible sugar. There are no constraints here except those defined by the notation, making the proof trivial to check.

Now, I will show that this trick can be played in reverse: Alpha theorem proving can be used to prove theorems for which semantics is governed by laws that is yet to be specified in a formal way.

Notice that in path semantics, there is no exhaustive definition of what path semantics is. One would then suspect that path semantics is not consistent. However, the surprising thing is that no exhaustive definition is needed!

Remember, the `α` function maps to *natural numbers*. While the way mathematical objects are mapped can be difficult to check for correctness, assigning a number to an object is a trivial operation to do, once you know which number it should be assigned to. You might ask: Why is this such a big deal?

The major insight of alpha theorem proving is that
the definition of the mathematically theory **does not matter**,
as long the α-*assignment* in any particular proof is consistent with
any **future development of the definition**!

For example, if I assign `α(x) = 1` and `α(y) = 2` in one proof, then after a while I come up with some insight that `α(x) ¬= α(y)`, which might not be obvious at first, I can simply go back and check:

```
α(x) ¬= α(y)
1 ¬= 2
true
```

While the reason for this inequality was not known until later, it had no bad impact on the existing proofs because they *already* satisfied the constraints that was to be discovered in the future.

This means alpha theorem proving permits a style of mathematical proofs that favors intuition, yet in a formal way, such that proofs can be checked later with future versions of path semantics.

However, there are some things worth to keep in mind:

- Alpha theorem proving only works on proofs of finite amount of expressions
- You are allowed to describe uncountable sets, as long you talk about them in a countable set
- It might be very hard to prove things about programs that generate new programs
- It might be very hard to prove things about programs that modify themselves

While this might seem like a severe limitation at first, it is not. Since the `α` function is part of path semantics itself, as long as you are able to define how uncountable sets are talked about in terms of a countable set using the `α` function, even meta-uncountable sets might be possible to ground, as long there is a countable proof somewhere that has a satisfied `α` function. Also, these proofs do not need to be found before you assume them, since you can assume they will be found sometime in the future or possible to be found given infinite computing power and time.

It is important that the `α` function remains uninterpreted at the start for every new proof. This is what makes it special from other functions. You can not assume that `α` function has a specific universal interpretation, otherwise path semantics can not be developed further. Since the language of path semantics is unbounded across infinite cardinal hierarchies, it means that the `α` function must remain uninterpreted, or “erased” sort of speak.

Notice that there is no requirement that the `α` function is the same for all observers. It only matters that there exists a way to assign mathematical objects to natural numbers, irrespective of the underlying nature of what these mathematical objects represent. If this is possible, then the proof has been proven and can be built upon without further mental burden. The shared knowledge lies within the language of path semantics, not in the semantics of the specific `α` function for a particular proof.

It might seem weird that a such thing is possible, but considering that path semantics only concerns itself about how one can talk about objects to distinguish their identities, it follows from how the `α` function works that everything that is said about the mathematical objects in question are relations that are relevant in a particular proof of countable cardinality. Otherwise, one must create a device able to communicate an infinite amount of information.

Alpha theorem proving solves a major milestone of the goals of path semantics, which is how to check path semantical proofs for soundness. I also avoided specifying any particular semantics of path semantics in general, which are subjects for future papers.