Proof Systems as Grammars

by Sven Nilsen, 2018

In this paper I represents a technique for constructing proof systems by building grammars.

The following language grammar for a proof system was constructed by starting with `true`:

true
A ∧ ¬A
“1+1=2” ∧ “1+1=3”

This language is able to make the following judgements:

“1+1=2” : true
“1+1=3” : false

Instead of thinking about `true` as a boolean value, think about it as a rule. This rule can be replaced by other rules, but only if the type of the rule permits it. In general, the type of the rule is of the form:

A → T

Where `A` is the input type for the next rule and `T` is the produced value.

For example, the language above was generated using the rules:

true => A ∧ ¬A	: (A → (= true)) × (A → (= false)) → (= true)
true => “1+1=2”	: str → (= true)
true => “1+1=3”	: str → (= false)

In closure form:

(= true)
\(tr : A → (= true), fa : A → (= false)) = \(a : A) = tr(a) ∧ ¬fa(a)
\(fa : str → (= false)) = \(a : str) = (= “1+1=2”)(a) ∧ ¬fa(a)
\(a : str) = (= “1+1=2”)(a) ∧ ¬(= “1+1=3”)(a)

When one has some rule that satisfies the trivial path of the first argument, it can be inserted.
The following arguments might depend on the previous one.

The trick of replacing `true` with some closure is that `true` can be generalized to $`true_n`$.
Instead of taking zero arguments, it can take many arguments, for all which it returns `true`.
All closures with this type that returns `true` is functionally equivalent.

When an argument takes a function of type `A → (= true)`, one can use recursion.
However, in the end one must have a parser that can work on strings.
So, the leaves in this grammar must be of type `str → B` where `B` is accepted by the parent rule.