A Quick Primer on TNT

Alexander Fraebel

April 14, 2021

1 Introduction

TNT crate for the Rust programming language that creates simple runtime validated proofs in Number Theory. This primer will explain the basics of using the crate.

The sections on Terms and Formulas explain how valid statements of TNT are formed. They mostly can be summarized by the Backus-Naur Form below. This precisely specifies Numbers, Variables, and Expressions and open Formulas. However quantified Formulas have additional restrictions that cannot be expressed as a context free grammar.

```
<num> ::= { "0" | "S" <num> }

<var> ::= { <lowercase_letter> | <var> "’" }

<arith> ::= { "+" | "*" }

<expr> ::= { <num> | <var> |
             "S" expr |
             "(" <expr> <arith> <expr> ")" }

<quant> ::= { "A" <var> ":" | "E" <var> ":" }

<logical> ::= { "&" | "|" | ">" }

<formula> ::= { <expr> "=" <expr> |
                <quant> <formula> |
                "~" <formula> |
                "[" <formula> <logical> <formula> "]" }
```

2 Terms

The grammar of TNT starts with three types: Variables, Numbers, and Expressions. Each of these implements the Term trait. The structs for these types all contain a String representing the underlying statement of TNT, the Variable struct also includes some additional information for speed of parsing. For brevity we will just refer to the contents of the struct. All Terms can be combined arithmetically. A few examples appear at the end of this section.

Numbers A valid Number consists of `0` preceded by `S` any number of times. These represent the natural numbers and the `S` symbol may be interpreted as the successor function. For convenience the Number type implements the methods .zero() and .one() which create `0` and `S0` automatically. The .random() method creates a random geometrically distributed Number.

Variables A valid Variable consists of any lowercase ASCII letter followed by any number of apostrophes. For instance `a` is a Variable as is `u”`. These stand for some unspecified natural number. The .random() method creates a Variable with a uniformly random letter and a geometrically distributed number of apostrophes.

Expression A valid Expression is a Number, Variable, an Expression preceded by `S`, or a arithmetic combination of two Expressions. The valid arithmetic operations are `+` and `*` and they must be parenthesized. Expressions may be arbitrarily complicated. For example `S(Sa’+0)`, `SSSSq`, `(SS0*(S(h’*S0)+j))`. The .random() method creates a random Expression of unbounded size.

3 Formulas

Formulas are well-formed formulas of the TNT language. The Formula enum has two variants: Simple and Complex. A Formula::Simple must be two Expressions with `=` between them such as `SS0=(a*b)`. In general a Formula can combine two Formulas with any of three binary logical operations, `&`, `|`, `>`. They may also be preceded with a tilde or with a quantification. Valid quantifications take the following form: start with `A` or `E`, then a Variable that does not appear within any quantification in the Formula, then `:`.

4 Deductions

Deductions are the centerpiece of the crate as they are required for using and checking the rules of inference. Internally the Deduction struct keeps a list of Formulas with some extra information. We will refer to the Formulas on this list as a "theorems" as they may be considered formally true within the TNT system of logic.

Making inferences with the Deduction is done through the methods provided below. Most of these methods take an index of a previous theorem as an argument. Methods never modify an existing theorem, they always create a new one.

All methods return a Result type with an explanation of the error if necessary. For the type and trait constraints see the full documentation.

`.add_axiom()` Adds the provided Formula to the list if it is in the axioms.

`.specification()` Clone the index, remove the universal quantification of the provided Variable, then change every occurrence of the Variable to the Term provided.

`.generalization()` Clone the index and add a universal quantification of the provided variable at the front.

`.existence()` Clone the index and add a existential quantification of the provided variable at the front.

`.successor()` Clone the index and prepend `S` to both sides of the equality.

`.predecessor()` Clone the index and remove `S` from both sides of the equality.

`.interchange_ea()` Clone the index and replace a negation of an existential quantification with a universal quantification of a negation.

`.interchange_ae()` Clone the index and replace universal quantification of a negation with a negation of an existential quantification with.

`.symmetry()` Clone the index and switch the sides of the equality.

.transitivity() Takes two indices. Creates a new theorem that is the the equality of the left side of the first with the right side of the second.

.supposition() Increases the depth by one step, creating a supposition block, and adds the provided Formula to the list.

.implication() Takes no arguments. Decreases the depth by one step then checks the previous supposition block and adds a theorem to the list that the first theorem implies the last theorem.

.induction() Takes a Variable and two indices. Adds a new theorem that is induction of the provided Variable on the provided base case and general case.