A Quick Primer on TNT

Alexander Fraebel

October 29, 2022

1 Introduction

TNT crate for the Rust programming language that creates simple runtime validated formal proofs in Number Theory. This crate adapts Typographical Number Theory from Chapter 8 of Douglas Hofstadter's *Gödel, Escher, and Bach.*

The first two sections, describing 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 Terms, however Formulas have additional restrictions that cannot be expressed as a context free grammar.

```
<var> ::= { <lowercase_letter> | <var> "'" }
<arith> ::= { "+" | "*" }
<term> ::= { "0" | <var> | "S" <term> |
             "(" <term> <arith> <term> ")" }
<quant> ::= { "A" <var> ":" | "E" <var> ":" }
<logical> ::= { "&" | "|" | ">" }
<formula> ::= { <term> "=" <term> | <quant> <formula> |
                "~" <formula> |
                "[" <formula> <logical> <formula> "]" }
```

2 Terms

The grammar of TNT starts with Terms. The Term type is an enumerable type with five variants: Zero, Variable, Successor, Sum, Product. The variants Zero and Variable are atomic while the others contains Terms. Simple Terms can be created using the enum's built in constructors. For more complex Terms is is suggested to use use the `Term::try_from()` method.

Examples of valid Terms are:

```
SSS0
(f''+((0*Sg)*a)
SS(S(b+h)*Sb)
```

The compact way that Terms are written simplifies their definition and implementation but can make them hard to read. Consequently a `.pretty_string()` method is provided that renders them in an easier to read format, spacing out arithmetic and changing ‘*‘ to ‘×‘.

```
SSS0
(f'' + ((0 × Sg) × a)
SS(S(b + h) × Sb)
```

3 Formulas

Formulas are well-formed formulas of the TNT language and are represented by an enum with seven variants: Equality, Universal, Existential, Negation, And, Or, Implies. The simplest Formula is Equality which contains two terms, all other variants contain one or more Formulas. The Universal and Existential variants are quantifications that, formally, must contain a Variable and a Formula. However due to limitations of the type system they instead contain a String (naming a Variable) and a Formula.

For constructing Formulas the `Formula::try_from()` method is suggested.

Examples of valid Formulas are:

```
Ac:[Ad:(d+Sc)=(Sd+c)>Ad:(d+SSc)=(Sd+Sc)]
Eb:(b*b)=a
Ab:Ac:[(SSb*c)=a>c=S0]
Ez':Sz'=0
(SS0*SS0)=SSS0
```

Notice that some of these Formulas have false or nonsensical interpretations. A Formula type only enforces the property of being well-formed, validity relies upon the Deduction struct.

To aid in interpretation of Formulas the `.to_english()` method is provided which translates the symbols to semi-readable English.

```
for all c, [for all d, (d + Sc) = (Sd + c)
   implies that for all d, (d + SSc) = (Sd + Sc)]
```

```
there exists b such that (b × b) = a

for all b, for all c, [(SSb × c) = a implies that c = S0]

there exists z’ such that Sz’ = 0

(SS0 × SS0) = SSS0
```

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. This list consists of theorems that are true within TNT and the set of axioms chosen. Using the `Deduction::peano()` creates a Deduction within axioms that correspond to Peano Arithmetic and thus to standard mathematics.

Making inferences with the Deduction is done through the thirteen 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.

`.specification(n, var_name, term` Eliminates universal quantification of Variables with the given name and replaces every instance of the Variable with the Term provided.

`.generalization(formula, var_name)` Remove every universal quantification of the provided Variable, then change every occurrence of the Variable to the Term provided.

`.existence(formula, var_name)` Clone the index and add a universal quantification of the provided variable at the front.

`.existence(n, var_name)` Add an existential quantification of a Variable with the provided name.

`.successor(n)` Prepend `S` to both sides of the equality.

`.predecessor(n)` Remove the first `S` from both sides of the equality.

`.interchange_ea(n, var_name, pos)` Replace the pos instance of negation of an existential quantification with the universal quantification of a negation.

`.interchange_ae(n, var_name, pos)` Replace the pos instance of universal quantification of a negation with the negation of an existential quantification with.

`.symmetry(n)` Switch the sides of the equality.

`.transitivity(n1, n2)` Create a new theorem that is the the equality of the left side of the first formula with the right side of the second formula.

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

`.implication()` Decrease the depth of the Deduction 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(var_name, base, general)` Adds a new theorem that is induction of the provided Variable on the base case and general case.