A Short Ride Through a Vast Machine

(A terse guide to System $\mathbf{T}^{++}$)

James Caldwell

January 1, 1992

Abstract

This short note is a compilation of all the logical and typing rules that comprise the system loosely referred to as $\mathbf{T}^{++}$. These rules were presented in the advanced programming languages course CS611 in the Fall of 1991 by Robert Constable. In principle, the class notes produced by students of CS611 contain all the rules presented here. I have pulled all the rules together in a uniform presentation and have added short remarks following each section.

1 Introduction

The system of rules informally known as $\mathbf{T}^{++}$ is based on Gödel's system $\mathbf{T}$ which (as presented by Girard in [2]) is the simply typed λ-calculus extended to include pairs, Peano arithmetic and Booleans. $\mathbf{T}^{++}$ is a further extension that includes the disjoint union type $(+)$, and the dependent product and dependent function types Σ and Π as well.

The Nuprl book [1] contains complete descriptions, including a section on the semantics, of all the rules presented here. It should be noted that the notation used there is the linear notation used in the Nuprl theorem prover and differs from the mathematical notation used here.

The two main properties of $\mathbf{T}^{++}$ are Strong Normalization and the fact that it characterizes the primitive recursive functions in all finite types. This class

can be viewed as a hierarchy of the following form.

Level	Types
$\vdots$	$\vdots$
L_{i+1}	$\{(L_j \to L_i) \mid j \leq i\} \cup \{(L_i \to L_j) \mid j \leq i\}$
$\vdots$	$\vdots$
2	$((\mathbf{N} \to \mathbf{N}) \to (\mathbf{N} \to \mathbf{N}))$, $((\mathbf{N} \to \mathbf{N}) \to \mathbf{N})$, $(\mathbf{N} \to (\mathbf{N} \to \mathbf{N}))$
1	$(\mathbf{N} \to \mathbf{N})$
0	$\mathbf{N}$

$\mathbf{T}^{++}$ should be contrasted with Plotkin's system **PCF** [3] which captures the general recursive functions in all finite types, i.e. it is Turing complete for functions $(N \to \mathbf{N})$.

1.1 Propositions as Types

The logic and typing rules are presented here in parallel to illustrate the Curry Howard isomorphism or notion *propositions as types*. Taken together, the logical rules characterize an intuitionistic or constructive logic (one of many.) Being constructive, the logical constructs have computational content which is reflected in the typing rules. Thus, propositions in the logic correspond to types in $\mathbf{T}^{++}$. Extending the comparison further, proofs of propositions correspond to programs or closed terms in $\mathbf{T}^{++}$. Looking at it the other way, each well typed closed term of $\mathbf{T}^{++}$ encodes a Natural Deduction proof of a proposition. Thus we can write $p \in P$ to mean p is a proof of proposition P or equivalently (by the isomorphism) to mean the p is a term of type P. For a somewhat more detailed account see the unpublished CS611 class notes or Thompson [4].

Each logical and typing construct has four kinds of rules. *Formation rules* determine the syntactic classes of formulas and types. So the formation rules inductively specify syntax. The *introduction rules* specify how introduction of instances of each construct may be justified. For obvious reasons, introduction rules are also sometimes called synthesis rules. The *elimination rules* (or analysis rules) specify how a construct can be decomposed and the *computation rules* specify how new constants are to be reduced.

1.2 Notational Conventions

To read the rules an understand them it must be realized that each statement above the horizontal line denotes a hypothesis of the rule and the statement below the line is the conclusion. If two or more hypotheses depend on each another they are stacked above one another, the lower ones depending on those above. In the logical formation rules $\mathcal{F}$ denotes the syntactic category of logical formulae.

Throughout these notes, depending on whether the rule is a logic rule or a typing rule, Γ denotes a list of premises or a *typing environment* (sometimes also called a *context.*) A typing environment Γ is a list of the form $x_1 : T_1, x_2 : T_2, \cdots, x_n : T_n$ where each x_i is a variable and each T_i is a type. We write $x : T$ in an environment to indicate that the variable x denotes an element of type T (or x denotes a proof of T;) that is, we can substitute any element of type T (any proof of T) for any free occurrence of x in the scope of the typing environment. We write $x \in T$ to indicate the membership of x in T is a proof obligation. As usual $\Gamma \vdash t \in T$ denotes provability of membership of t in T based on the assumptions encoded by the typing environment Γ.

As usual $P[a/x]$ denotes the substitution of a for all free occurrences of x in P.

The notation used for the dependent type constructors Σ and Π is not standardized. The variant used here, $\Sigma x : A.B$ and $\Pi x : A.B$, is essentially the notation used by Martin-Löf. The Nuprl book adopts `x:A#B` and `x:A->B` which can be written in standard ascii. Other notation includes $\Sigma(A; x.B)$ and $\Pi(A; x.B)$.

2 Conjunction and the Cartesian Product Type

$$\frac{A \in \mathcal{F} \quad B \in \mathcal{F}}{(A \wedge B) \in \mathcal{F}}\ (\wedge F) \qquad \frac{A \text{ is a type} \quad B \text{ is a type}}{(A \times B) \text{ is a type}}\ (\times F)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash (A \wedge B)}\ (\wedge I) \qquad \frac{\Gamma \vdash p \in A \quad \Gamma \vdash q \in B}{\Gamma \vdash \langle p, q \rangle \in (A \times B)}\ (\times I)$$

$$\frac{\Gamma \vdash (A \wedge B)}{\Gamma \vdash A}\ (\wedge E_l) \qquad \frac{\Gamma \vdash \langle p, q \rangle \in (A \times B)}{\Gamma \vdash p \in A}\ (\times E_l)$$

$$\frac{\Gamma \vdash (A \wedge B)}{\Gamma \vdash B}\ (\wedge E_r) \qquad \frac{\Gamma \vdash \langle p, q \rangle \in (A \times B)}{\Gamma \vdash q \in B}\ (\times E_r)$$

$$\frac{\Gamma \vdash (A \wedge B) \quad \Gamma, A, B \vdash C}{\Gamma \vdash C}\ (\wedge E)$$

$$\frac{\Gamma \vdash p \in (A \times B) \quad \Gamma, u{:}A, v{:}B \vdash t \in T}{\Gamma \vdash \mathtt{spread}(p; u, v.t) \in T}\ (\text{spread})$$

$$\mathtt{spread}(\langle t_1, t_2 \rangle; u, v.t) \rightarrow t[t_1, t_2/u, v]$$

The essence of the isomorphism here is that conjunction corresponds to a pair in the type theory. A pair containing p and q as its left and right components is denoted $\langle p, q \rangle$. A proof of $(A \wedge B)$ is a pair containing a proof of A and a proof of B. Thus introduction rule $(\times I)$ is just a pair construction and the elimination rules project out the left or right components of the pair. The pair decomposition operator is **spread**. The computation rule specifies that **spread** decomposes a pair into its left and right elements and binds them to variables which may occur free in the term t. Note that we can define the standard pair projection functions as follows:

$$fst = \lambda p.\mathtt{spread}(p; u, v.u) \qquad snd = \lambda p.\mathtt{spread}(p; u, v.v)$$

3 Disjunction and the Disjoint Union Type

$$\frac{A \in \mathcal{F} \quad B \in \mathcal{F}}{(A \vee B) \in \mathcal{F}}\ (\vee F) \qquad \frac{A \text{ is a type} \quad B \text{ is a type}}{(A + B) \text{ is a type}}\ (+F)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash (A \vee B)}\ (\vee I_l) \qquad \frac{\Gamma \vdash a \in A}{\Gamma \vdash \mathtt{inl(a)} \in (A + B)}\ (+I_l)$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash (A \vee B)}\ (\vee I_r) \qquad \frac{\Gamma \vdash b \in B}{\Gamma \vdash \mathtt{inr(b)} \in (A + B)}\ (+I_r)$$

$$\frac{\Gamma \vdash (A \vee B) \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee E)$$

$$\frac{\Gamma \vdash d \in (A + B) \quad \Gamma, u{:}A \vdash t_1 \in T \quad \Gamma, v{:}B \vdash t_2 \in T}{\Gamma \vdash \mathtt{decide}(d; u.t_1; v.t_2) \in T}\ (+E)$$

$$\begin{array}{l} \mathtt{decide}(\mathtt{inl}(\mathtt{a}); u.e; v.f) \rightarrow e[a/u] \\ \mathtt{decide}(\mathtt{inr}(\mathtt{b}); u.e; v.f) \rightarrow f[b/v] \end{array}$$

The key point here is that a disjunction on the logic side corresponds to a disjoint union type constructor in type theory. The symmetries of these rules with the conjunction rules should be noted. Here there are two introduction rules and one elimination rule. For conjunction we have one introduction rule and two elimination rules. Note that `inl` and `inr` are injections mapping elements into the left or right components of a disjoint union and are used in the introduction rules $(\vee I_l)$ and $(\vee I_r)$. A common mistake is to assume that $a{:}A \vdash a \in (A + B)$. Clearly a disjoint union $(A + B)$ is not the same type as A whence `inl` and `inr`. `decide` is a linear representation of a two way case split as can be seen from $(+E)$ and the computation rules.

4 Implication and the Function Type

$$\frac{A \in \mathcal{F} \quad B \in \mathcal{F}}{(A \Rightarrow B) \in \mathcal{F}}\ (\Rightarrow F) \qquad \frac{A \text{ is a type} \quad B \text{ is a type}}{(A \to B) \text{ is a type}}\ (\to F)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash (A \Rightarrow B)}\ (\Rightarrow I) \qquad \frac{\Gamma, x{:}A \vdash b \in B}{\Gamma \vdash \lambda x.b \in (A \to B)}\ (\to I)$$

$$\frac{\Gamma \vdash (A \Rightarrow B) \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow E) \qquad \frac{\Gamma \vdash f \in (A \to B) \quad \Gamma \vdash a \in A}{\Gamma \vdash (fa) \in B}\ (\to E)$$

$$(\lambda x.b)a \to b[a/x]$$

The essence of the isomorphism in this case is that implication in the logic corresponds to the function type in type theory. A proof of $(A \Rightarrow B)$ is a function that maps a proof of A to a proof of B. Rule $(\Rightarrow I)$ is an inference rule encoding the Deduction theorem and rule $(\to I)$ characterizes λ abstraction. Thus, an application of the deduction theorem corresponds to λ-abstraction on the type theory side. The rule $(\Rightarrow E)$ is just *modus ponens* and corresponds to a function application. The computation rule for function application is just β-reduction.

5 Absurdity, Negation and the Void Type

$$\frac{}{\bot \in \mathcal{F}}\ (\bot F) \qquad \frac{}{\texttt{void} \text{ is a type}}\ (\texttt{void}F)$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ () \qquad \frac{\Gamma \vdash x \in \texttt{void}}{\Gamma \vdash \texttt{any(x)} \in T}\ (\texttt{anyI})$$

$$\neg A \equiv A \Rightarrow \bot$$

$\bot$ denotes logical absurdity or false. A type A is inhabited if and only if the proposition corresponding to A is true. This informally justifies the isomorphism between $\bot$ and `void`, the empty type. `any(x)` is any type. The identification of $\neg A$ with $A \Rightarrow \bot$ also makes sense. Looking at the type theory side, if there is an inhabitant of A, say $a \in A$, then A is true and $\neg A$ is not. Thus $\neg A$ is true if and only if A is `void`.

6 Existential Quantification and the Σ–Type

$$\frac{A \in \mathcal{F} \quad B \in \mathcal{F}}{\Gamma \vdash \exists x{:}A.B \in \mathcal{F}}\ (\exists F) \qquad \frac{A \text{ is a type} \quad B \text{ is a type}}{\Sigma x{:}A.B \text{ is a type}}\ (\Sigma F)$$

$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash B[a/x]}{\Gamma \vdash \exists x{:}A.B}\ (\exists I) \qquad \frac{\Gamma \vdash a \in A \quad \Gamma \vdash b \in B[a/x]}{\Gamma \vdash \langle a, b\rangle \in \Sigma x{:}A.B}\ (\Sigma I)$$

$$\frac{\Gamma \vdash \exists x{:}A.B \quad \Gamma, u{:}A, v{:}B[u/x] \vdash C}{\Gamma \vdash C}\ (\exists E)$$

$$\frac{\Gamma \vdash p \in \Sigma x{:}A.B \quad \Gamma, u{:}A, v{:}B[u/x] \vdash t \in C}{\Gamma \vdash \mathtt{spread}(p; u, v.t) \in C}\ (\Sigma E)$$

The Σ-type is also sometimes written $x{:}A \times B(x)$ reflecting its relationship with the Cartesian product type and the relationship between conjunction and the existential quantifier.

Σ types are also known as dependent product types. Informal justification for this interpretation can be seen by examining the formation and introduction rules. By examination of the formation rule we see that if $x : A$ then $B[a/x]$ denotes a type, (by the proposition as types interpretation it is also a proposition.) Turning to the introduction rule we see that the pair $\langle a, b\rangle$ is in the type $\Sigma x{:}A.B$ exactly when it can be shown that $a \in A$ and $b \in B[a/x]$. That is, for any arbitrary element a of type A, b is an element of type $B(a)$. The *type* of the second element may depend on the first element.

The canonical example of a dependent type is `date` datatype having month and day components denoted `M` and `D` respectively. If we let $[1..n]$ denote the type containing the natural numbers 1 through n then `date`,`M` and `D` types can be specified as follows:

$$\begin{aligned}
&\mathtt{date} = \Sigma m{:}\mathtt{M}.\mathtt{D}\\
&\mathtt{M} = [1, .., 12]\\
&\mathtt{D} = (\lambda m.\ \mathtt{if}\ m \in \{1,3,5,7,8,10,12\}\ \mathtt{then}\ \ [1, ..31]\\
&\qquad\qquad \mathtt{elseif}\ m \in \{4,6,9,11\}\ \mathtt{then}\ \ [1, .., 30]\\
&\qquad\qquad \mathtt{else}\ [1, .., 28])
\end{aligned}$$

By rule (ΣI), $\langle m, d\rangle \in \mathtt{date}$ if and only if $m \in \mathtt{M}$ and $d \in \mathtt{D}(m)$.

7 Universal Quantification and the Π-Type

$$\frac{A \in \mathcal{F} \quad B \in \mathcal{F}}{\forall x{:}A.B \in \mathcal{F}}\ (\forall F) \qquad \frac{A \text{ is a type} \quad B \text{ is a type}}{\Pi x{:}A.B \text{ is a type}}\ (\Pi F)$$

$$\frac{\Gamma, x{:}A \vdash B}{\Gamma \vdash \forall x{:}A.B}\ (\forall I) \qquad \frac{\Gamma, a{:}A \vdash b \in B[a/x]}{\Gamma \vdash \lambda(x.b) \in \Pi x{:}A.B}\ (\Pi I)$$

$$\frac{\Gamma \vdash \forall x{:}A.B \quad \Gamma \vdash a \in A}{\Gamma \vdash B[a/x]}\ (\forall E) \qquad \frac{\Gamma \vdash f \in \Pi x{:}A.B \quad \Gamma \vdash a \in A}{\Gamma \vdash (fa) \in B[a/x]}\ (\Pi E)$$

$$(\lambda x.b)a \rightarrow b[a/x]$$

The Π-type is a generalized function type and is sometimes written $x{:}A \rightarrow B$ reflecting this relationship.

Thinking about the computational content of a universally quantified statement yields the intuition behind the Π-type. It is essentially the same justification as that given for the function type. That is, if b is of type $B[a/x]$ then for all elements a of type A, the term $(\lambda x.b)$ will map a to an element of type $B[a/x]$. Looking at it from the programs as proofs point of view this is exactly what we expect. A proof of $\forall x{:}A.B$ must map proofs of x of A to proofs b of $B[a/x]$.

8 N and Peano Arithemtic

$$\frac{}{\mathbf{N} \text{ is a type}}\ (\mathbf{N}F)$$

$$\frac{}{0 \in \mathbf{N}}\ (\mathbf{N}I_0) \qquad \frac{\Gamma \vdash n \in \mathbf{N}}{\Gamma \vdash \mathtt{succ}(\mathbf{n}) \in \mathbf{N}}\ (\mathbf{N}I_{\mathtt{succ}})$$

$$\frac{\Gamma \vdash b \in T(0) \quad \Gamma, u{:}N, v{:}T(u) \vdash t \in T(succ(u))}{\Gamma, x{:}N \vdash \mathtt{ind}(x; b; u, v.t) \in T(x)}\ (\mathbf{N}E)$$

$$0 \to 0$$
$$\mathtt{succ}(a) \to \mathtt{succ}(a)$$
$$\mathtt{ind}(0; b; u, v.t) \to b$$
$$\mathtt{ind}(succ(n); b; u, v.t) \to t[n/u, ind(n; b; u, v.t)/v]$$

These rules of the type theory characterize Peano arithmetic. Note that rule $(\mathbf{N}E)$ uses a parameterized type T which we have not used before. This deserves more dicussion which I don't have time for now.

From the computation rules it can be seen that the induction form `ind` is just a linearized form of the primitive recursion schema over dependent types with b corresponding to the base case and t corresponding to the induction case.

The logical rule for mathematical induction arises from the rule $(\mathbf{N}E)$.

$$\frac{\Gamma \vdash T(0) \quad \Gamma, u{:}N, T(u) \vdash T(succ(u))}{\Gamma, x{:}N \vdash T(x)}\ (\text{induction})$$

9 Subject Reduction

$$\frac{\Gamma \vdash t \in T \quad T =_{\mathbf{T}^{++}} T'}{\Gamma \vdash t \in T'}$$

We need this rule to preserve subject reduction for $\mathbf{T}^{++}$. $=_{\mathbf{T}^{++}}$ denotes the the equivalence induced by the reduction rules of $\mathbf{T}^{++}$ presented above.

References

[1] Robert L. Constable et. al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[2] Jean-Yevs Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[3] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.

[4] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.