Demonstrating the Undecidability and Incompleteness of Peano Arithmetic with a Higher Order Logic Theorem Prover

Robert T. Bauer
Dr. Raymond J. Toal

12/3/1997

Department of Electrical Engineering
and Computer Science
College of Science and Engineering
Loyola Marymount University

Acknowledgments

The work presented in this paper could not have been accomplished without Dr. Ray Toal's help and encouragement; however, it is important to recognize that Loyola Marymount University has been very kind to allow me to pursue a course of study dominated by theoretical topics in Computer Science, Metamathematics, Logic, and Distributed Computing.

Dr. Ray Toal, Dr. Phil Dorin, and Dr. Tony Karrer have each provided me with an example of excellence, dedication, and scholarship; to each of them, I offer my deepest and most humble respect.

In no particular order, I would like to acknowledge the following persons: Lisa and Mike Malgeri for their friendship and Ray Bala II for showing that excellence needs no loudspeaker. I especially thank Albert James Ingram (age twelve) who stated "A number is a thing that you use to count other things." I hope that one day he reads this work and comes to understand both the limitations and significance of his statement.

Finally, a personal statement: In the course of this work, I have walked in the footsteps of great intellectual achievement and have been protected from the vastness of confusion by the shadow of genius. I have chased the diagonal, touched the edge of infinity, and experienced the awe of understanding. I cannot claim the clarity of vision nor the depth of understanding of Galileo, Cantor, Dedekind, Frege, Russell, Church, Turing, Godel, or Milner; however, I am satisfied that this work will help others on their journey to understanding.

Table of Contents

Abstract

Godel's Incompleteness Theorem (GIT) expresses the undecidability of any formal system that is a recursively axiomatizable extension of First Order Peano Arithmetic. The goal of this work is to demonstrate a proof of GIT using the Higher Order Logic (HOL) Theorem Prover. Since HOL is based on a variant of Church's Type Theory, we provide a comprehensive background in type theory and HOL. We then present a backwards demonstration and proof of GIT in type theory. Finally we take initial steps towards the demonstration of GIT in the HOL Theorem Prover.

1. Introduction

Godel's First Incompleteness Theorem stands as the most important 20th Century landmark in Mathematical Logic. In 1931, Kurt Godel [9] demonstrated that any reasonably powerful formal system of axioms and inference rules contained true sentences that could not be proved – i.e., the formal system was incomplete. Godel constructed a true sentence that could not be proved and whose complement (by definition, false) could not be refuted. Within the formal system, this sentence is undecidable. In a relatively short paper, Kurt Godel crushed all hopes of using *algorithms* to correctly identify whether a given formal statement was a theorem in a given formal theory of mathematics.

Following Smullyan [18], consider a computer program that examines sentences. If the sentence is true, the computer program prints the sentence. Given a finite alphabet, we could generate all of the sentences of one symbol, then two symbols, etc., – and our program would only print those that were true. Given enough time, our program enumerates all truths within our language. To test whether a sentence represents a true statement, we simply count the number of symbols appearing in the sentence, index to the appropriate part of the print-out (or wait for the sentence to be printed, confirming the old adage that "truth comes to those who wait") and then find the sentence being examined or determine that it is not (will never be) printed. If the sentence appears on the print-out, it represents truth, otherwise it represents falsehood.

Let P mean "printable" in the following sense. Suppose that some symbol, say Z has the value "false," then our program will not print Z; however, P Z is also not printable because it is not true that Z is printable (P Z = false). Let ~ represent negation and note that ~ P Z will be printed (because P Z is not printable) and also that ~ Z will be printed. Let "(" and ")" be used to group symbols, e.g., ~P (P Z) – a true statement, meaning that P Z is not printable. Other (true) groupings include P ~(P Z) and P (~P Z). Let N(X) stand for X(X). So, for example, P N (P N) is printable if N(P N) is true and N(P N) is P N(P N); therefore, if we interpret N(P N) as true, then P N(P N) is printed, otherwise it is not. Now consider the sentence ~P N(~P N). Suppose we interpret N(~P N) as true, then ~P N(~P N) is not printed; however, N(~P N) is ~P N(~P N) – resulting in a true sentence that is not printed. Suppose we interpret N(~P N) as false. Then ~P N(~P N) is printed; however, N(~P N) is ~P N(~P N) – resulting in a false sentence that is printed. If our program is consistent, it cannot print a false sentence; therefore, we must admit that while our machine only prints "true" sentences, there are true sentences that it does not print.

A formal system is said to be complete if and only if every "true" sentence is derivable from the axioms. It is said to be sound if and only if every sentence derivable from the axioms is "true." The formal system is said to be consistent if only if every sentence derivable from the axioms implies that its complement is not derivable from the axioms. The sentence above, shows that our program is either incomplete (a true sentence not printed) or inconsistent (a false sentence that is printed). Now consider the sentence P N(~P N).

Suppose N(~P N) is true, then P N(~P N) is printed; however N(~P N) is ~P N(~P N) – which if true, means that the sentence we just printed is false, and if false, means we should not have printed the sentence. Suppose that N(~P N) is false, then P N(~P N) is not printed; but N(~P N) is ~P N(~P N) and if false, means that we did not print a true sentence. P N(~P N) is an example of an undecidable sentence: A formal system is said to be decidable if and only if, for any sentence we can determine whether it or its complement is derivable from the axioms.

The goal of this project is to formulate Godel's First Incompleteness Theorem within the framework of the Higher Order Logic (HOL) theorem prover. The significance of this effort

is entirely technical; yet, it lays the foundation for further metamathematical work within the HOL system. In preparation of this effort, we present a Type-theoretic version of Godel's First Incompleteness Theorem. Whereas the traditional presentation generally proceeds as a forward proof, our presentation follows the spirit of HOL and presents a backward proof. As we work backwards through the proof, we identify the requirements for new tacticals and/or the selection of existing tacticals.

Although Andrews [1] does a very fine job of describing type theory including the development Godel's First and Second Incompleteness Theorems, we relied on Smullyan [18], Nagel [16], Shankar [17], Davis [7], and Godel [9] to fully understand and appreciate Godel's work. Throughout the backwards proof, we fill in many details that will not be found in Andrews; thus, this work may be considered a definitive, type-theoretic exposition of Godel's First Incompleteness Theorem.

The remainder of this work is organized as follows: Section 2 provides an overview of Higher Order Logic and especially the definition of new types and a detailed description of tactics. Section 2 also provides an overview of Andrew's type theory (Q) including the development of natural numbers (Peano Axioms) and Axioms of Infinity. We also compare and contrast Peano Arithmetic as expressed in HOL. In Section 3, we illustrate a type-theoretic proof of Godel's Incompleteness Theorem. Section 4 outlines Godel's Incompleteness in terms of HOL. Lastly, Section 5 presents our conclusions and provides ideas and directions for further research.

2. Type Theory

Church's λ-calculus is a type free theory about functions as rules rather than as graphs. Dirichlet is usually credited with the important idea that functions could be considered as sets of pairs of argument and value (extending Descartes' notion of pairs), but the λ-calculus retains the notion of functions as rules to stress the computational aspects of functions [2]. λ-calculus was developed to provide a general theory about functions and their manipulation. It was hoped that it would also help provide a foundation for (parts of) mathematics.

While λ-calculus was successful in describing functions, all attempts to provide a foundation in mathematics failed. Typed λ-calculus was introduced in order to prevent the various paradoxes (e.g., Russell's) from demonstrating the inconsistencies of these mathematical systems. All "type" systems, including typed λ-calculus, allow new types to be specified:

- The set of types are inductively defined as follows.
    1. There exists a base type (in Church's original paper, the base types consisted of propositional and individual corresponding to Andrew's individuals and truth (boolean); in HOL, only individuals are in the base type, all other types are derived).
    2. If σ, τ are types, then $\sigma \rightarrow \tau$ (meaning a function mapping type σ to type τ is also a type.
- Thus, all functions, variables, and constants (e.g., +) are strongly typed. For example, the "plus" constant for adding two objects of type number is specified as $+_{\sigma\rightarrow\sigma}$ where σ means number.

$(\alpha\beta)$ means $\beta\rightarrow\alpha$. Also, $(\alpha\beta\delta)$ stands for $\delta\rightarrow\beta\rightarrow\alpha$. Note that types are left associative. Thus, $\delta\rightarrow\beta\rightarrow\alpha$ and $\delta\rightarrow(\beta\rightarrow\alpha)$ are equivalent; however, $(\delta\rightarrow\beta)\rightarrow\alpha$ denotes a function that takes (as an argument) functions of the type $\delta\rightarrow\beta$ and returns an α. $\delta\rightarrow\beta\rightarrow\alpha$, on the other hand, denotes a function that takes a δ and returns a function that takes a β and returns an α.

In the remainder of this section, we discuss the type systems of HOL and Andrew's system Q. We conclude this section with a comparison of formalized numbers in HOL and Q.

2.1. HOL

Higher Order Logic (HOL) is called higher order because:

- Variables are allowed to range over functions and predicates;
- Functions can take functions as arguments and yield functions as results; and
- The notation of the λ–calculus can be used to write terms that denote functions.

In HOL, each variable has an associated logical type that specifies the kind of values it ranges over. Sets and even recursive structures (lists and trees) can be represented in HOL by extending the syntax of types. This is done by first defining these new types in terms of already existing types and then deriving properties about these new types by formal proof. The idea is that formal proof will guarantee that adding a new type will not introduce inconsistency.

The syntax of HOL includes terms that correspond to conventional predicate calculus – P x expresses the proposition that x has property P. R(x,y) means that relation R holds between

x and y. Examples of P include **Man** ***Socrates*** to express the notion that ***Socrates*** is human and <(3,5) to denote 3 < 5.

In HOL, `T` indicates truth, `F` falsity, `~` negation, `\/` disjunction, `/\` conjunction, `==>` implication, `=` equality, `!` universality, `?` existential, `@` hilbert's e operator, `=>` conditional, and `?!` unique existence. For example, a theorem of primitive recursion states that for any equation written inductively (base and induction statement), there exists a unique recursive function computing the same value:

```
|- ?F .!x f .((F f)0 = x) /\ !n .(F f)(n+1) = f ((F f) n)
```

For any variable x, for any function f: function F, which takes a function as an argument and yields a function as a result, has the following properties. Given the base of the induction (f 0 = x), F takes f and 0 as arguments and yields x; thus, F is equivalent in the base case. And (/\) for all n (!n), F takes f and n+1 as arguments and computes the same value as F taking f and n as arguments, the result being applied to f (recursively).

HOL is a mechanized, interactive, proof-assistant that has been primarily used to reason about the correctness of digital hardware; however, much of what has been developed in HOL for hardware verification – the theory of arithmetic, is also fundamental for other applications...

HOL supports secure theorem proving by representing its logic in a ML, a strongly-typed functional programming language. Propositions and theorems of the logic are represented by ML abstract data types and interaction with the theorem prover takes place by executing ML procedures that operate on values of these data types. Users can write arbitrarily complex programs to implement proof strategies and because the logic is represented using ML abstract data types, the user-defined proof strategies are guaranteed to perform only valid logical inferences.

2.1.1. Defining New Logical Types

The use of types in higher order logic eliminates the potential for the inconsistency that comes with allowing higher order variables. For example, Russell's Paradox: P x = ~ x x, then substitute P for x, leading to P P = ~ P P, is avoided using typed variables.

In HOL, three steps are needed to define a new type:

1. find an appropriate subset of an existing type to represent the new type;
2. extend the syntax of logical types to include a new type symbol, and using a *type definition axiom* to relate this new type to its representation; and
3. deriving from the type definition axiom and the properties of the representing type a set of theorems that serves as an 'axiomatization' of the new type.

In the first step, a model for the new type is given by specifying a set of values – this is done by defining a predicate P on an existing type such that the set of values satisfying P denotes the properties that the new type is expected to have. In the second step, a new type constant or type operator is added to the logic – this *type definition axiom* serves to relate values of the new type to the corresponding values of the existing type that represents them.

In the last step, a collection of theorems is proved that abstractly characterizes the new type. Here "abstractly" means that essential properties of the new type are stated without reference to the way the values are represented; we use the term 'axiomatized' because the collection of theorems are derived from the definition of the subset predicate defining the set of values for the new type and from the *type definition axiom*. This last step gives a

consistency proof of the axioms (theorems?) for the new type by showing that there is a model for them.

Suppose that α is an existing type of the logic and P:$\alpha\rightarrow$bool is a predicate on values of type α that defines the subset for the new type. Since the semantics of the hilbert e operator require all types to denote non-empty sets, it must be the case that "|- ?x:α.P x" and we must prove it before the type definition axiom can be added to the system.

The type definition axiom asserts the existence of a representation function that maps a value of the new type to the value of the "base" type that represents it:

$$\vdash ?f:\beta\rightarrow\alpha\ .(!a_1\ a_2\ .f\ a_1 = f\ a_2 \supset a_1 = a_2)\ /\backslash\ (!r\ .P\ r = (?a\ .r = f\ a))$$

This axiom states that function f which maps from the new type (β) to the "base" type (α) is one-to-one and onto the subset defined by P. Note that new operators can be defined in a similar fashion. If t is of type (α,β), then P:t(α,β)$\rightarrow$bool is the desired predicate and if ?x.P x, then a new binary type operator (α,β)*op* is asserted:

$$\vdash ?f:(\alpha,\beta)op\rightarrow t(\alpha,\beta)\ .(!a_1\ a_2\ .f\ a_1 = f\ a_2 \supset a_1=a_2)/\backslash(!r\ .P\ r =(?a\ .r = f\ a))$$

To formulate the abstract axioms required in step three, it is convenient to have logical constants which in fact denote the isomorphism (and its inverse). A representation function that maps values in the new type to the value in the "base" type is defined:

$$\vdash REP = @f\ .(!a_1\ a_2\ .f\ a_1 = f\ a_2 \supset a_1 = a_2)\ /\backslash\ (!r\ .P\ r = (?a\ .r = f\ a))$$

Once REP is defined, the e-operator can be used to defined the inverse abstraction function ABS: base_type $\rightarrow$ new_type:

$$\vdash !r\ .ABS\ r = (@a\ .r = REP\ a)$$

From these definitions, it follows that ABS is the left inverse of REP and for REP is the left inverse of ABS:

$$\vdash !a\ .ABS(REP\ a) = a$$

$$\vdash !r\ .P\ r = (REP(ABS\ r) = r)$$

2.1.1.1.A Simple Type Definition

The simplest and smallest type possible in HOL is the type constant *one* denoting a set containing exactly one element. To represent one, any singleton subset of an existing type will do. Since T (truth within the type bool) is one of the two elements that are represented in the type bool, we can use the predicate λb:bool .b denoting the identity function on bool. This set is non-empty, |-?x. (λb.b)x follows from |- (λb.b)T which is equivalent to |- T.

Now that we have established that λb.b specifies a non-empty set of booleans, the type constant one can be defined:

$$\vdash ?f:one\rightarrow bool\ .(!a_1\ a_2\ .f\ a_1 = f\ a_2 \supset a_1 = a_2)\ /\backslash\ (!r\ .(\lambda b.b)r = (?a\ .r = f\ a))$$

and we specify f as REP_one:*one*$\rightarrow$bool. The axiomatization of the type *one* consists of a single theorem:

$$\vdash !f:\alpha\rightarrow one\ .!g:\alpha\rightarrow one\ .(f = g)$$

This theorem says that all functions mapping values of type α to values of type *one* are equal. Thus, there is only one value of type *one* (otherwise $f \neq g$ at least once!). Note that we have specified an essential property of the type without reference to how the type is represented.

Of course we need to prove the theorem we have just stated:

```
|- !a1 a2 .REP_one a1 = REP_one a2 ⊃ a1 = a2
|- !r .r = (?a .r = REP_one a)
```

Specializing r in the second equation (if it is true for all r, then it is certainly true for any r) to `REP_one(f x)` results in:

```
|- REP_one(f x) = (?a .REP_one(f x) = REP_one a)
```

Note that the right-hand side is T, since our new type is non-empty, so that REP_*one* a maps to T any a which is of type *one*. The theorem simplifies to `|- REP_one(f x)` and similar reasoning yields `|- REP_one(f x)`. Then since equal things are equal, we have `|- REP_one(f x) = REP_one(g x)`. From this and the first equation which states that REP_one is one-to-one, we have `|- f x = g x` and therefore `|- !f g .(f = g)`.

Finally we note that once we prove the axiom about *one*, we can easily demonstrate that there is only one value of type *one*. Here's the definition of a constant called **one**:

```
|- one = @x:one .T
```

From the previously proved axiom about *one*, we have "`|- λx:α .v = λx:α .one`" – resulting in "`|- v = one`". Since v is not free in **one**, we generalize:

```
|- !v:one .v = one
```

stating that every value v of type *one* is equal to the constant **one**.

2.1.2. Tactics, Tacticals and Backward Proofs

HOL uses sequents to express logical constructs. A sequent is a pair (Γ, t) where Γ is a finite set of formulae called the set of assumptions and t is a single formula called the conclusion. A deductive system D is a set of pairs (L,(Γ, t)) where L is a possibly empty list of sequents. A sequent follows from a set of sequents, Δ, if and only if there exist sequents (Γ_1, t_1) , ..., (Γ_n, t_n) such that:

1. $(\Gamma, t) = (\Gamma_n, t_n)$; and
2. for all i such that $1 \le i \le n$:
   - $(\Gamma_i, t_i) \in \Delta$; or
   - $(L_i\ (\Gamma_i, t_i)) \in D$ for some list of L_i of members of $\Delta \cup \{(\Gamma_1, t_1)\ , \ldots, (\Gamma_n, t_n)\}$.

The sequence (Γ_1, t_1) , ..., (Γ_n, t_n) is called a proof of (Γ, t) from Δ with respect to (model) D. As is usual, HOL specifies its deductive system as a number of (schematic) rules of inference:

- Assumption Introduction – Assume t results in the sequent "t |– t".
- Reflexivity – "|– t = t".
- Beta–conversion – "|– t (λx .t)u = t[u/x]"; substitute u for x.
- Substitution – "Γ_1 |– $t_1 = u_1$, ..., Γ_n |– $t_n = u_n$, Γ |– $t[t_1, \ldots, t_n] \rightarrow \Gamma_1 \cup \ldots \cup \Gamma_n \cup$ Γ |– $t[t_1, \ldots, t_n]$".
- Abstraction – "Γ |– t = u → Γ |– (λx .t) = (λx .u)" provided x is not free in Γ.
- Type Instantiation – "Γ |– t → Γ |– $t[\alpha_1, \ldots, \alpha_n / \beta_1, \ldots, \beta_n]$" where $t[\alpha_1, \ldots, \alpha_n / \beta_1, \ldots, \beta_n]$ is the result of substituting, in parallel, the types α for the type variables β with two conditions:
  1. None of the type variables, β, occur in Γ; and

2. No distinct variables in t become identified after the instantiation. For example, let F(v) stand for $\exists t .(t \neq v)$. Clearly, by F(t) we do not mean the incorrect formula $\exists t .(t \neq t)$, but rather $\exists z .(z \neq t)$.

- Discharging an Assumption – "$\Gamma \vdash t \rightarrow \Gamma - \{u\} \vdash u \rightarrow t$".
- Modus Ponens – "$(\Gamma_1 \vdash t_1 \rightarrow t_2 , \Gamma_2 \vdash t_1) \rightarrow \Gamma_1 \cup \Gamma_2 \vdash t_2$".

All provable statements of HOL may be derived from the rules of inference given above and the five axioms listed below:

- $\vdash \forall b .(b = T) \vee (b = F)$
- $\vdash \forall b_1 b_2 .(b_1 \rightarrow b_2) \rightarrow (b_2 \rightarrow b_1) \rightarrow (b_1 = b_2)$
- $\vdash \forall f_{\alpha \rightarrow \beta} .(\lambda x . f x) = f$
- $\vdash \forall P_{\alpha \rightarrow bool} x .P x \rightarrow P (@ P)$
- $\vdash \exists f_{ind \rightarrow ind} .One\_One f \wedge \neg(Onto f)$

Note, these axioms are extended through the development of new theories (c.f., Section 3.5 where we detail the theory of numbers in HOL).

A theorem is a sequent that follows from the empty set of sequents; thus, a theorem is the last element of a proof from the empty set of sequents. A forward proof proceeds from the "axioms" to the theorem being proved. A backwards proof proceeds from the theorem back to the axioms.

Although one can perform forward proofs in HOL, the HOL theorem prover was designed to support backward (goal directed) proofs. A *tactic* in HOL is an ML function that when applied to a goal reduces it to a list of sub-goals and a justification function that maps a list of theorems to a theorem. For example, suppose the "goal" was to prove a simple goal such as "$T \wedge \neg F$" is straight-forward.

In HOL there is a tactic called CONJ_TAC. It takes a conjunction, such as the goal specified above, and breaks it into two sub-goals and a justification function. The justification function takes a list of "values" and generates a theorem. Let Truth stand for "$\vdash T$". Note that "Truth" has no assumptions. Let Not_False stand for "$\vdash \neg F$". Again note that there are no assumptions. The use of the justification function for CONJ_TAC is shown below:

CONJ_TAC_JUSTIFICATION [Truth, Not_False] $\Rightarrow \vdash T \wedge \neg F$

Tactics are ML functions, so HOL users are free to create their own tactics. The ML type inference system guarantees soundness and the HOL system actually carries out the functions specified to ensure that proofs are evaluated.

A tactic solves a goal if it reduces the goal to the empty set of sub-goals. To do this requires at least one tactic that maps a goal to the empty sub-goal list. ACCEPT_TAC is a function that maps a goal to the empty list of sub-goals.

Tacticals differ from tactics. A tactical is an ML function that returns a tactic or list of tactics as a result. The most common example is the tactical ORELSE. If S and T are tactics, S ORELSE T evaluates to a tactic that applies S unless that fails. If S fails, ORELSE applies T. The tactical FIRST is similar, except that it applies the first tactic in a list of tactics that succeed.

2.2. *Andrew's System Q*

Andrew's type theory, like all type theoretic systems, is based on Church's typed λ–calculus where every individual, variable, function, and predicate has a type. For example, the function + that operates on pairs of integers to yield their sum is denoted by $+_{(\sigma\sigma\sigma)\sigma}$ where σ is the type of integers. In Andrew's type theory, the type o denotes the type of truth values (i.e., boolean) so any function of type (oα) identifies a set of elements of type α or not type α.

2.2.1. Primitive Basis of Q_0

α, β, ..., ζ, etc., are variables that range over type symbols. Type symbols are defined inductively:

> o is a type symbol denoting the type of truth values[1].
>
> ι is a type symbol denoting the type of individuals.
>
> If α and β are type symbol, then (αβ) is a type symbol denoting the type of functions from elements of type β to elements of type α (e.g., f:β→α).

$Q_{o\alpha\alpha}$ denotes the binary relation of equality. If A_o is a wff of type o in which the variable y_ι occurs free, and if there is a unique element y_ι of which A_o is true, we denote that element by $[\iota y_\iota A_o]$ – meaning the y_ι such that A_o. In type theory, $[A_\alpha = B_\alpha]$ abbreviates $[Q_{o\alpha\alpha} A_\alpha B_\alpha]$ and the truth value True abbreviates $[Q_{ooo} = Q_{ooo}]$. Falsity, the truth value of False abbreviates $[\lambda x_o T] = [\lambda x_o x_o]$ (we define false as the statement "every individual of type truth is true," since we can easily demonstrate the false truth value).

The logical constants of Q_0 are equality and selection:

- $Q_{((o\alpha)\alpha)}$; and
- $\iota_{(\iota(o\iota))}$

By wff_α , we mean a wff of type α. We write A_α , B_α , etc., as variables ranging over wff's of type α. Wff's are defined inductively:

1. Any primitive variable or constant of type α is a wff_α ;
2. $[A_{\alpha\beta}B_\beta]$ is a wff_α ; and
3. $[\lambda x_\beta A_\alpha]$ is a $wff_{\alpha\beta}$.

2.2.2. Axioms of Q_0

1. $g_{oo}T_o \wedge g_{oo}F_o = !\chi_o .g_{oo}\chi_o$ – meaning that truth and falsehood are the only truth values.
2. $[X_\alpha = Y_\alpha] \supset .h_{o\alpha}X_\alpha = h_{o\alpha}Y_\alpha$ – the results of function application are equal when the arguments are equal.
3. $(f_{\alpha\beta} = g_{\alpha\beta}) = !X_\beta .f_{\alpha\beta}X_\beta = g_{\alpha\beta}X_\beta$ – so that equal functions produce equal results.
4. $[\lambda x_\alpha B_\beta]A_\alpha = B_\beta$ where B_β is a primitive constant or variable distinct from x_α – showing that the entire range is mapped by the entire domain.
5. $[\lambda x_\alpha x_\alpha]A_\alpha = A_\alpha$ – λ identity function abstraction.

---

[1] Unfortunately Andrew's chose not to refer to this set as the set of boolean values. In previous sections we refer to "truth" values as boolean; however, since this is Andrew's theory, we keep his terminology.

6. $[\lambda x_\alpha .B_{\beta\gamma}C_\gamma]A_\alpha = [[\lambda x_\alpha B_{\beta\gamma}]A_\alpha]\ [[\lambda x_\alpha C_\gamma]A_\alpha]$ – substitution is simultaneous, even in the presence of composition.
7. $[\lambda x_\alpha .\lambda y_\gamma B_\delta]A_\alpha = [\lambda y_\gamma.[\lambda x_\alpha\ B_\delta]A_\alpha]$ where y_γ is distinct from x_α and from all variables in A_α.
8. $[\lambda x_\alpha .\lambda x_\alpha B_\delta]A_\alpha = [\lambda x_\alpha\ B_\delta]$
9. $\iota_{(o\iota)}[Q_{o\iota}\ y_\iota] = y_\iota$

Q_0 has only one rule of inference: From C and $A_\alpha = B_\alpha$ to infer the result of replacing one occurrence of A_α in C by an occurrence of B_α, provided that the occurrence of A_α in C is not an occurrence of a variable immediately preceded by λ.

2.2.3. Elementary Logic in Q_0

The class of propositional wffs is the smallest class of wff_o containing T_o , F_o , every variable of type o and if it contains A and B also contains:

- $[\sim A]$
- $[A \wedge B]$
- $[A \vee B]$
- $[A \rightarrow B]$
- $[A = B]$

Recall that the operators in the above sentential forms are abbreviations for the primitive constructions given in the previous section.

Variables of type o may be assigned truth values; thus if A represents a wff, then $\mathcal{V}_\varphi$ A refers to the value of the wff A given some assignment of truth values to the variables of A. We now give the denotation of the logical basis of Q:

1. $\mathcal{V}_\varphi\ T_o = T$ (truth)
2. $\mathcal{V}_\varphi\ F_o = F$ (falsehood)
3. $\mathcal{V}_\varphi\ p_o = \varphi\ p_o$ (the "value" of a propositional variable is the value it is assigned)
4. $\mathcal{V}_\varphi\ (\sim A) = T$ if $\mathcal{V}_\varphi\ (A = F$ and $\mathcal{V}_\varphi\ (\sim A) = F$ if $\mathcal{V}_\varphi\ (A) = T$
5. $\mathcal{V}_\varphi\ [A \wedge B] = T$ if $\mathcal{V}_\varphi\ (A) = T$ and $\mathcal{V}_\varphi\ (B) = T$ otherwise $\mathcal{V}_\varphi\ [A \wedge B] = F$
6. $\mathcal{V}_\varphi\ [A \vee B] = T$ if $\mathcal{V}_\varphi\ (A) = T$ or $\mathcal{V}_\varphi\ (B) = T$ otherwise $\mathcal{V}_\varphi\ [A \vee B] = F$
7. $\mathcal{V}_\varphi\ [A \rightarrow B] = T$ if $\mathcal{V}_\varphi\ (A) = F$ or $\mathcal{V}_\varphi\ (B) = T$ otherwise $\mathcal{V}_\varphi\ [A \rightarrow B] = F$
8. $\mathcal{V}_\varphi\ [A = B] = T$ if $(\mathcal{V}_\varphi\ (A) = \mathcal{V}_\varphi\ (B))$ otherwise $\mathcal{V}_\varphi\ [A = B] = F$

Although we do not prove the soundness and completeness of Q with respect to elementary logic[2], we state the main conclusion: if A is a tautology ($\mathcal{V}_\varphi$ (A) = T for all truth-value assignments φ), then A is a theorem of Q: Tautology(A) $\rightarrow$ $\vdash$ A.

[2] Andrew [1] is the obvious source for information regarding Q; however, [12] is a very good source for metatheoretic results concerning propositional and first order logic.

2.2.4. Formalized Number Theory in Q_0

2.2.4.1. Definition of natural numbers

Two sets are said to be equivalent[3] if the members of each set can be mapped one-to-one. The cardinal number of a set is "that which is common" to all equivalent sets. Thus, if x is equivalent to y, the cardinality of x must equal the cardinality of y: $x \approx y \rightarrow \underline{\underline{x}} = \underline{\underline{y}}$. It is interesting to note that set theory has no convenient way to express the notion of equivalence. In Q_0 equivalence is expressed:

$$[\lambda p_{o\beta}\lambda q_{o\alpha}\exists s_{\alpha\beta}\bullet\forall x_{\beta}[\, p_{o\beta}x_{\beta} \rightarrow q_{o\alpha}\bullet s_{\alpha\beta}x_{\beta}] \wedge \forall y_{\alpha}\bullet q_{o\alpha}y_{\alpha} \rightarrow \exists_1 x_{\beta}\bullet p_{o\beta}x_{\beta} \wedge y_{\alpha} = s_{\alpha\beta}x_{\beta}]$$

which has the type $((\beta\rightarrow o)\rightarrow(\alpha\rightarrow o))\rightarrow o$ and which "asserts" the existence of a function s that establishes a one-to-one and onto relationship between the sets p and q. If the sets are not equivalent, s does not exist so that the lambda-expression is false.

Letting $E_{((\beta\rightarrow o)\rightarrow(\alpha\rightarrow o))\rightarrow o}$ stand for the equivalence relation, cardinal numbers are expressed:

$$[\lambda u_{o(o\beta)}\exists p_{o\beta}\bullet u_{o(o\beta)} = E_{o(o\beta)(o\beta)}p_{o\beta}]$$

with the type $((\beta\rightarrow o)\rightarrow o)\rightarrow o$. As defined earlier, a cardinal number is that which is common to all equivalent sets. If u is a cardinal number, then it represents such a commonality as follows: u is the cardinal number for some set (of β individuals) if there exists some other set of β individuals equivalent to the one referenced by u.

Having defined cardinal numbers, we wish to define ordinal numbers (so that we can say, for example, that one number is less than another). In traditional set theory, we would define zero as a symbol that associates with the empty set (i.e., $0 \equiv \{\}$). The symbol one would represent the set containing the empty set ($1 \equiv \{\ \{\}\ \}$). Clearly, only sets with a single element are equivalent to 1. Inductively, we define the set n to be the set containing the set zero, 1, ..., n–1 ($n \equiv \{\ \{\}$, "1", "2", ... "n–1"}). Where "1" means equivalence class of sets containing exactly one element.

That 2 is greater than 1, is argued as follows: The ordinal number 2 is a symbol associated with a set of sets equivalent to the set containing the empty set and the sets equivalent to 1. Since the set represented by 2 contains the set represented by 1, the set represented by 2 is larger than 1.

In type theory, the concept of ordinality is much simpler. The successor function in type theory is constructed around the notion that the cardinal number associated with some set A is a successor to the cardinal number associated with a set B, if by removing one element from set A we obtain a set, say A' with the same cardinality as that of B. With this notion, type theory easily expresses the notion of order.

The symbol σ is used to represent the equivalence classes of sets of individuals $(\iota\rightarrow o)\rightarrow o$ and may be interpreted as "number" throughout the remainder of this paper. The equivalence class of sets with zero members is used to represent zero:

0_{σ} stands for $Q_{o(o\iota)(o\iota)}[\lambda x_{\iota}\, F_{o}]$

The successor function rewrites a lambda expression to generate the successor:

$S_{\sigma\sigma}$ stands for $[\lambda n_{o(o\iota)}\lambda p_{o\iota}\exists x_{\iota}\bullet p_{o\iota}x_{\iota} \wedge n_{o(o\iota)}[\lambda t_{\iota}\bullet t_{\iota}\neq x_{\iota} \wedge p_{o\iota}t_{\iota}]]$

Consider $S_{\sigma\sigma}S_{\sigma\sigma}0_{\sigma}$ (representing the number 2):

---

[3] The set-theoretic notion of equivalence is very different from ordinary usage of the term. Although one would like to use equinumerous instead of equivalence neither numbers nor equality has been defined!

$$[\lambda p_{o\iota}\exists x_\iota \bullet p_{o\iota}x_\iota \wedge [\lambda q_{o\iota}\ \exists z_\iota \bullet q_{o\iota}\ z_\iota \wedge Q_{o(o\iota)(o\iota)}\ [\lambda w_\iota F_o][\ \lambda u_\iota \bullet u_\iota \neq z_\iota \wedge q_{o\iota}u_\iota]][\lambda t_\iota \bullet t_\iota \neq x_\iota \wedge p_{o\iota}t_\iota]]$$

$$[\lambda p_{o\iota}\exists x_\iota \bullet p_{o\iota}x_\iota \wedge [\exists z_\iota \bullet [\lambda t_\iota \bullet t_\iota \neq x_\iota \wedge p_{o\iota}t_\iota]z_\iota \wedge Q_{o(o\iota)(o\iota)}\ [\lambda w_\iota F_o][\ \lambda u_\iota \bullet u_\iota \neq z_\iota \wedge [\lambda t_\iota \bullet t_\iota \neq x_\iota \wedge p_{o\iota}t_\iota]u_\iota]]]$$

$$[\lambda p_{o\iota}\exists x_\iota \bullet p_{o\iota}x_\iota \wedge [\exists z_\iota \bullet z_\iota \neq x_\iota \wedge p_{o\iota}z_\iota] \wedge Q_{o(o\iota)(o\iota)}\ [\lambda w_\iota F_o][\ \lambda u_\iota \bullet u_\iota \neq z_\iota \wedge u_\iota \neq x_\iota \wedge p_{o\iota}u_\iota]]$$

The set p such that x is in p, z is in p, u is in p and the set u is empty and there is an x in p which is not the same as any z in p and neither z nor x are equivalent to the empty set. Clearly this represents the set with exactly two (distinct) elements. More precisely, the lambda expression represents the characteristic function of the equivalence relation: It is true precisely when applied to argument that is a set with two distinct elements.

The type theoretic representation of the natural numbers:

$N_{o\sigma}$ stands for $[\lambda n_\sigma\ \forall p_{o\sigma} \bullet [\ p_{o\sigma}0_\sigma \wedge \forall x_\sigma \bullet p_{o\sigma}x_\sigma \rightarrow p_{o\sigma}S_{o\sigma}x_\sigma\ ] \rightarrow p_{o\sigma}n_\sigma]$

If p is true for zero and if being true for some number it is also true of that number's successor, we say p is a property of numbers. If we consider the set of all such properties of numbers, then the set of numbers may be defined as the set of things that have all of those properties.

As discussed previously, cardinal numbers represent that which is common to equivalent sets. In this sense, when we write the symbol 2, we are using a sign to represent that notion of "in-common." Indeed, it would very inconvenient to write Q[λx F] for zero or SSQ[λx F] for 2. Since natural numbers as defined above are simply equivalence classes and because every equivalence class has a cardinality, then every natural number is a cardinal number:

$\text{Finite}_{o(o\iota)}$ stands for $[\lambda p_{o\iota}\ \exists n_\sigma \bullet N_{o\sigma}n_\sigma \wedge n_\sigma p_{o\iota}]$

For any set, p, there exists a number n that represents "how many" things are in the set.

2.2.4.2. Peano Axioms

The axiomatization of the natural numbers is usually accomplished through the Peano Axioms:

(a) Zero is a natural number.

(b) If n is a natural number so is Sn (Sn is the successor of n, often written n').

(c) If Sm=Sn, then m = n.

(d) There is no number n for which Sn = 0.

(e) Axiom of complete induction:

> If a property P of the natural numbers satisfies the following two conditions, then P holds for every natural number:
>
> (1) P holds for 0.
>
> (2) For every natural number n, if P holds for n, then P holds for Sn.

Expressed in type theory, the Peano Axioms become:

(a) $\vdash N_{o\sigma}\ 0_\sigma$

Proof. By definition of N.

(b) $\vdash \forall x_\sigma . N_{o\sigma}\ x_\sigma \rightarrow N_{o\sigma} . S_{o\sigma}x_\sigma$

Proof. Suppose the converse. Then, $\exists y\ .\neg N\ y \rightarrow N\ S\ y$, asserting the existence of an equivalence relation upon the successor but not the predecessor. By definition of the successor function the equivalence of the predecessor is established.

(c) $\vdash \forall n_\sigma\ \forall m_\sigma\ .S_{\sigma\sigma}n_\sigma = S_{\sigma\sigma}m_\sigma \rightarrow n_\sigma = m_\sigma$

Proof. Suppose the converse – that S n = S m and $n \neq m$. Since "=" stands for $Q_{\sigma\sigma\sigma}$ and because the successor function S uses Q, we have the curious phenomena that the Q on the lhs is somehow different from the Q on the rhs.

(d) $\vdash \forall n_\sigma\ .S_{\sigma\sigma}n_\sigma \neq 0_\sigma$

Proof. Consider the definition of the successor expression and the definition of zero. Zero represents all sets that are equivalent to the empty set. The successor function specifies that the successor set can be made equivalent to the predecessor set by removing an individual from the successor set. If zero is the successor of some number, then it is possible to remove an individual from the empty set.

(e) $\vdash \forall p_{\sigma\sigma}\ .[p_{\sigma\sigma}0_\sigma \wedge \forall x_\sigma\ .p_{\sigma\sigma}x_\sigma \rightarrow p_{\sigma\sigma}\ .S_{\sigma\sigma}x_\sigma] \rightarrow \forall x_\sigma p_{\sigma\sigma}x_\sigma$

Proof. The definition of N says that if a property is valid for the equivalence relation with the empty set and if it is true for some equivalence relation and the successor of the equivalence relation, then it is true for a specific equivalence relation. The definition of complete induction says that given any property of natural numbers, if that property is valid for zero and given any natural number, that property is valid on the successor, then the property is valid for all natural numbers.

2.2.4.3. Axioms of Infinity

Infinity has played a very fundamental role in the modern development of mathematics and computer science. Intuitionism attacked the entire notion of infinity by claiming that exhaustive searches fail with infinite domains and that proof-by-contradiction cannot yield a constructive demonstration of existence in an infinite domain. Formalism in mathematics was a reaction to intuitionism – the assumption was that infinities were convenient idealizations and that conclusions based on infinities could be shown to be correct. Out of this formalism grew metamathematics where the whole scope of mathematics became the object of algorithms (symbols and syntactic rules for forming sentences and proofs).

Thus, from Dedekind, Cantor, Peano, Ferge, Hilbert, Whitehead & Russell, and finally to Kurt Godel, we travel from abstract notions of infinity to a concrete demonstration of undecidability which in its own way expresses an entirely different kind of infinity. Yet, it is almost startling to learn that Galileo was aware of the one-one correspondence between the set of natural numbers and the set of squares of the natural numbers.

Following Dedekind, it is still convenient to use the characteristic property that any infinite set is equivalent to at least one of its proper subsets. Finite sets can be defined as any set that cannot be placed in one-one correspondence with any of its proper subsets. The axiom used in type theory, on the other hand, is one that is a bit more natural: there is an irreflexive partial ordering of individuals with respect to which there is no maximal element:

$$\exists r_{o\iota\iota} \forall x_\iota \forall y_\iota \forall z_\iota [\exists w_\iota\ r_{o\iota\iota}x_\iota w_\iota \wedge \neg r_{o\iota\iota}x_\iota x_\iota \wedge .r_{o\iota\iota}x_\iota y_\iota \rightarrow .r_{o\iota\iota}y_\iota z_\iota \rightarrow r_{o\iota\iota}x_\iota z_\iota]$$

When formalizing natural numbers, the axiom of infinity is not needed except to show:

$$\vdash \forall n_\sigma\ \forall m_\sigma\ .S_{\sigma\sigma}n_\sigma = S_{\sigma\sigma}m_\sigma \rightarrow n_\sigma = m_\sigma$$

2.3. *Comparing Formalized Numbers in HOL and Q_0*

HOL formalizes numbers as a theory that is derived from the theory of individuals. The theory of individuals is derived from the theory of booleans. In this section we quickly recap the theory of individuals (there are an infinite number of them) and the theory of booleans. We then compare and contrast how numbers are expressed in these two type-theoretic systems.

The theory of booleans has only implication ($\rightarrow$), equality (=), choice (which uses the symbol @ for Hilbert's ε-operator). Hilbert's ε-operator has already been introduced; however, we augment that introduction by noting that an HOL metatheorem states:

$$[\exists x .t\ x] = T \rightarrow [t\ @x .t\ x] = T$$

Our final remark concerning Hilbert's ε-operator is that if !x .t x=F, then @x .t x denotes an unspecified value (of the proper type). According to the HOL documentation, @x looks suspicious, but is very useful for naming things one knows to exist but for which one has no name!

The Hilbert ε-operator is equivalent to the Axiom of Choice (select the value of x such that x has the desired property) which when reformulated in HOL becomes [@x .P x].

With only three logical constants, here is how HOL defines boolean expressions and abbreviates the "standard" boolean connectives:

- T stands for $((\lambda x .x) = (\lambda x .x)) - Q$ defines T and F as primitives in the sense that T and F are the (only) values that may be assigned to boolean variables. So, in HOL, we have a precise notion of the value True.
- The binder ! (the HOL representation of $\forall$) stands for $[\lambda P .P = [\lambda x\ T]]$.
- The binder ? (the HOL representation of $\exists$) stands for $[@x .\lambda P .P\ x = T]$
- F stands for [!b .b]
- Negation (~) stands for $[\lambda b .b \rightarrow F]$
- And ($\wedge$) stands for $[\lambda x\ \lambda y .!z .(x \rightarrow (y \rightarrow z)) \rightarrow z]$
- Or ($\vee$) stands for $[\lambda x\ \lambda y .!z .(x \rightarrow z) \rightarrow ((y \rightarrow z) \rightarrow z)]$
- Equal (=) stands for $[\lambda x\ \lambda y .(x \rightarrow y) \wedge (y \rightarrow x)]$
- Unique existence (?!) $[\lambda P .?\ P \wedge !x .!y .P\ x \wedge P\ y . \rightarrow .x = y]$

The axioms related to theory of booleans:

- $!b .(b = T) \vee (b = F)$
- $!x .!y .(x \rightarrow y) \rightarrow (y \rightarrow x) \rightarrow (x = y)$
- $!f .(\lambda x .f\ x) = f$
- $!P\ x .P\ x \rightarrow P(@\ P)$

Other constants of the theory bool include One_One, Onto, and Onto_Subset, etc., some of these have already been discussed in regard to how new types are defined within HOL.

The theory of individuals extends the two element type boolean. If there are x distinct elements of type α and y distinct elements of type β, then there are y^x distinct elements of type $\alpha \rightarrow \beta$. Since there an infinitely many numbers, there is no hope of constructing a representing type for numbers from bool and implication ($\rightarrow$).

Individuals are a primitive type in HOL that contain infinitely many distinct elements. The axiom of infinity used in HOL is of a mapping from the class of individuals to the class of individuals such that the mapping is a function (One_One) but not Onto (does not map the entire range):

$$?F{:}ind \to ind\ .One\_One\ F \wedge \neg(Onto\ F)$$

This axiom is derives the natural numbers as follows:

The representing type for zero is

$$Zero\_Rep = @y{:}ind\ .!x\ .\ \neg(y = Suc\_Rep\ x)$$

meaning zero is the "value" of the representing type (individuals) such that this value is not the value of the representing type for the successor of any other individual.

Then,

$$!x\ .\sim (Zero\_Rep = Suc\_Rep\ x)$$

So that, $1 = Suc\_Rep\ (Zero\_Rep)$, $2 = (Suc\_Rep(Suc\_Rep(Zero\_Rep)))$, etc.

The proceeding demonstrates that in HOL numbers are represented by the subset of individuals (ind) consisting of Zero_Rep and all elements of the form $Suc\_Rep^n\ Zero\_Rep$. If we could define the function Is_Num as $[\lambda x\ .(x = Zero\_Rep) \vee ?n\ .x = Suc\_Rep^n\ Zero\_Rep]$ it would be very convenient; however, the definition assumes that n is a number. In HOL

Let Is_Num stand for $[\lambda x\ .!P\ .(P\ Zero\_Rep) \wedge (!y\ .P\ y \to P(Suc\_Rep\ y)) \to P\ x]$

Thus, we have

Is_Num Zero_Rep; and

$!x\ .Is\_Num\ x \to Is\_Num(Suc\_Rep\ x)$

Therefore, the representing type for num is the class of individuals. The subset predicate is Is_Num. The subset is non-empty since Is_Num Zero_Rep. The representation function is Rep_Num: num → ind.

Finally, in a manner reminiscent of our derivation of the Peano postulates in type theory, we develop them (Peano postulates) for HOL.

Define the abstract for numbers as Abs_Num = Inverse Rep_Num. Then,

- $0 = Abs\_Num\ Zero\_Rep$
- $Suc = Abs\_Num(Suc\_Rep\ (\ Rep\_Num))$

The first axiom, zero is a natural number follows by definition: Is_Num Zero_Rep;

The second axiom, if n is a number, then the successor of n is a number, follows from the definition of Suc.

The third axiom, if the successor of m equals the successor of n, then m and n are equal, can be proved based on the definition of One_One: $[\lambda f{:}\alpha \to b\ .!x\ .!y\ .(f\ x = f\ y) \to (x = y)]$.

The fourth axiom, zero is not the successor of any number, is the definition of zero in HOL.

The fifth axiom, complete induction, also follows from the definition of Is_Num:

$$[!P{:}num \to bool\ .P\ 0 \wedge (!m\ .P\ m \supset P\ (Suc\ m)) \supset !m\ .P\ m]$$

3. Godel's First Incompleteness Theorem

3.1. Godel's Sentence G

Previous sections have introduced the reader to basic concepts, notation, etc., of type theory and higher order logic. In this section, we start with an explanation of Godel's incompleteness theorem and work backwards through the proof. Our approach here is a bit unusual in that we begin with a type-theoretic version of Godel's sentence G and use it to introduce syntax and semantics:

Let G stand for $[\lambda x_\sigma\ \forall y_\sigma\ \neg \mathrm{Proof}^S_{\sigma\sigma\sigma}\ [\ \text{“[“} * x_\sigma * [\mathrm{Num}_{\sigma\sigma}\ x_\sigma] * \text{“]”}\ ]\ y_\sigma]_{\sigma\to o}$

Numbers have type σ and booleans have type o. Proof is a function that takes two numbers and returns a boolean value. The superscript S represents that *Proof* is specific to some formal system called S. The function Proof works as follows:

- Every sentence in the language is comprised of a sequence of one or more symbols from the alphabet of the language.
- Each sentence can be assigned a unique number called the Godel number of the sentence. We can determine whether an arbitrary number is a Godel number and we can recover the sentence encoded by the Godel number.
- Each unique series of sentences has a unique Godel number.
- A "proof" is a series of sentences with the following properties:
    1. Each sentence in the series either begins or ends a subseries in the sentence.
    2. Whether an arbitrary sentence is in the series is decidable.
    3. Each sentence in the series is either an axiom of the formal system or is the result of applying a rule of inference to one or more earlier sentences in the series.
    4. The last sentence in the series represents that which is being proved.
- Proof is true if and only if the second argument represents a "proof" whose last sentence is equivalent to the first argument.

The function proof contains the very powerful notion that an algorithm exists that can determine whether an arbitrary number represents the Godel encoding of a valid proof within the formal system being considered (in this case S). Indeed, Godel demonstrated just such an algorithm for Whitehead & Russell's Principia Mathematica.

The sequence of symbols "[" * x_σ * [$\mathrm{Num}_{\sigma\sigma}\ x_\sigma$] * "]" is quite complex: "[" represents the Godel number associated with the symbol [. Note, we use this notation quite often: "$\forall x$ .Suc $x \neq 0$" means the Godel number associated with the sentence enclosed in quotes. Concatenation is an operation of juxtaposition denoted by *. The semantics of concatenation are straightforward: "A" * "B" must be equivalent to "AB".

Num is a tricky function to understand. In a previous section we described the formalization of numbers within type theory (and Higher Order Logic). Generally, a number is represented as so many successors to zero; thus, our common notion of the number 5 is expressed as Suc Suc Suc Suc Suc 0 or equivalently SSSSS0 and sometimes (set-theoretic) 0"""". The function Num transforms the series of symbols into a single number-symbol. In the literature, one finds a difference between 5 and Num 5. 5 is some

symbol (whose Godel encoding does not have to be the number 5) whereas Num 5 is a symbol representing the concept of the number 5.

Consider the number 1003. It is comprised of a series of 4 symbols. We write 1003 because it is more convenient than writing SSSS...S0. However, Num 1003 is not 1003 (there are 1004 symbols in the representation!). It is possible, though, that some sentence has the Godel number 1003, e.g., "$\forall x = x$" $\equiv$ 1003 where 1003 is a single number-symbol. To avoid confusion, "$\forall x = x$" represents the Godelization of the sentence; Num "$\forall x = x$" represents the number associated with the Godelization. Note: Num "1003" $\neq$ 1003 and that Num 1003 = 1003, since 1003 is a single number-symbol whereas "1003" is the Godel number associated with the series '1' '0' '0' '3' of four symbols.

Getting back to sentence G, consider application to "G"; "G" represents the Godelization of G:

$$G\text{ "G"} \equiv [\lambda x_o\ \forall y_o\ \sim Proof_{ooo}^{S}\ [\text{ "["} * x_o * [Num_{oo}\ x_o] * \text{"]"}\ ]\ y_o]_{o\to o}\ \text{"G"}$$

$$G\text{ "G"} \equiv [\forall y_o\ \sim Proof_{ooo}^{S}\ [\text{ "["} * \text{"G"} * [Num_{oo}\ \text{"G"}] * \text{"]"}\ ]\ y_o]_{o\to o}$$

If G "G" is a theorem, then "G" * [Num_{oo} "G"] must be the last sentence of some valid proof. So if we assume $\vdash$ G "G" meaning that G "G" is a theorem, then we have

$$\vdash [\forall y_o\ \sim Proof_{ooo}^{S}\ [\text{ "["} * \text{"G"} * [Num_{oo}\ \text{"G"}] * \text{"]"}\ ]\ y_o]_{o\to o}$$

Since $\vdash$ G "G", then $\exists$n .Proof ["[" * "G" * [Num_{oo} "G"] * "]"] n

Specialize y to n:

$$\vdash [\sim Proof_{ooo}^{S}\ [\text{ "["} * \text{"G"} * [Num_{oo}\ \text{"G"}] * \text{"]"}\ ]\ n]_{o\to o}$$

Simplify:

$$\vdash [\sim T_o]$$

$$\vdash F_o$$

If we derive a falsehood, as we have done, then our system is inconsistent; rejecting inconsistency, we must accept $\sim\vdash$ G "G"; so that G "G" is not a theorem (i.e., cannot be proved).

If G "G" is not a theorem, it is possible that ~G "G" is a theorem. Assume $\vdash$ ~G "G", then

$$\vdash \sim [\forall y_o\ \sim Proof_{ooo}^{S}\ [\text{ "["} * \text{"G"} * [Num_{oo}\ \text{"G"}] * \text{"]"}\ ]\ y_o]_{o\to o}$$

Then, noting the equivalence, $\sim \forall \sim \equiv \exists$, we have

$$\vdash [\exists y_o\ Proof_{ooo}^{S}\ [\text{ "["} * \text{"G"} * [Num_{oo}\ \text{"G"}] * \text{"]"}\ ]\ y_o]_{o\to o}$$

If there existed a "proof", then G "G" would be a theorem, therefore we reject $\vdash$ ~G "G" and accept $\sim\vdash$ ~G "G".

Finally, we inquire as to the "truth" of G "G". Suppose G "G" were false, then there would exist some proof (as G "G" asserts that for all y, there is no proof); however, we have established that the existence of such a proof leads to inconsistency. Hence, G "G" is true (but not provable). Thus, with sentence G "G" we have demonstrated a true sentence that cannot be proved and with ~G "G" a false sentence that cannot be refuted.

Having worked through the top level of Godel's First Incompleteness Theorem, we now introduce some relevant definitions and concepts that have not been introduced:

Num, *, and Proof are representable[4] in Q^-. Then, if the formal system $\mathcal{A}$ is any recursively axiomatized[5] pure extension of Q^-, Num, *, and Proof are representable in $\mathcal{A}$. Let $\mathcal{A}$ be an extension of Q^-:

1. $\mathcal{A}$ is *consistent* iff $\sim \vdash_{\mathcal{A}} F_o$
2. $\mathcal{A}$ is ω–*consistent* iff there is no closed wff $B_{o\sigma}$ of $\mathcal{A}$ such that $\vdash_{\mathcal{A}} \exists y_\sigma B_{o\sigma} y_\sigma$ and for each natural number k, $\vdash_{\mathcal{A}} \sim B_{o\sigma} k$. Another way of stating this is that the formal system $\mathcal{A}$ is ω–*inconsistent* if there is a formula $F(\omega)$ in one free variable ω, such that the sentence $\exists\omega F(\omega)$ is provable, yet all the sentences F(0), F(1), ..., F(n), ... are refutable. A ω–*inconsistent* cannot be correct (only true sentences are provable), but it may be consistent. A system that is not ω–*inconsistent* is ω–*consistent* – meaning that whenever the sentence $\exists\omega F(\omega)$ is provable, then for at least one number n, F(n) is not refutable.
3. $\mathcal{A}$ is *complete* iff for each sentence A_o of $\mathcal{A}$, $\vdash_{\mathcal{A}} A_o$ or $\vdash_{\mathcal{A}} \sim A_o$
4. $\mathcal{A}$ is ω–*complete* iff $\vdash_{\mathcal{A}} \forall y_\alpha B_{o\sigma} y_\sigma$ for each closed wff $B_{o\sigma}$ of $\mathcal{A}$ such that $\vdash_{\mathcal{A}} B_{o\sigma} k$ for each natural number k.
5. $\mathcal{A}$ is *essentially incomplete* iff $\mathcal{A}$ is consistent and recursively axiomatized and every consistent recursively axiomatized pure extension of $\mathcal{A}$ is incomplete.

3.2. Representing Proof

Consider the following demonstration of Modus Ponens:

(1) Assume t1

(1.1) t1 $\vdash$ t1

(2) Assume t1 $\rightarrow$ t2

(2.1) t1 $\rightarrow$ t2 $\vdash$ t1 $\rightarrow$ t2

(3) Conclude (Modus Ponens) $\vdash$ t2

(3.1) t1, t1 $\rightarrow$ t2 $\vdash$ t2

Now consider the same demonstration written in a functional form:

Modus_Ponens(Assume(t1),Assume(t1 $\rightarrow$ t2)) = $\vdash$ t2

Godel's idea was that checking a proof could be done by an algorithm; and even more efficient, the proof itself could be written as algorithm (e.g., program). But perhaps, the most important result is that Godel precisely defined the concept of proof and then showed that it was recursive. Godel's definition of a recursive number-theoretic function extended the concept of general recursion to include the notion that a proof is also recursive: If a proof is a finite series of sentences such that the last sentence in the series is the conclusion (i.e., the sentence proved by the proof) and where each sentence is either an axiom or the result of applying a rule of inference to one or more earlier sentences, then clearly proof is recursive (indeed, it is obvious that is recursively enumerable).

---

[4] A function, F, is said to represent a number set A, $n \in A \leftrightarrow \vdash F(n)$. In type theory: !n. A n = $\vdash$ F n.

[5] Godel's definition of recursively axiomatized is "Function ψ is called recursive, if there exists a finite series of number-theoretic functions $\psi_1, \psi_2, \psi_3, \ldots, \psi_n$, which ends in ψ and has the property that every function ψ_k of the series is either recursively defined by two of the earlier ones, or is derived from any of the earlier ones by substitution or is a constant or the successor function."

With this background, let us delve into the details of the function proof. Informally, we may define proof as $[\lambda m_{\sigma},n_{\sigma} .\exists$ a proof P^0, ..., P^K such that m = "P^K" and n = "P^0, ..., P^K"$]_{\sigma\sigma\sigma}$ – however, we need a more precise definition:

$$[\lambda m,n \ .(m = E(n) \ \mathcal{GL} \ n)_o \wedge$$

$$(\forall k \leq E(n))[Axiom(k \ \mathcal{GL} \ n) \vee (\exists i < k)(\exists j < k) \ Rule(i \ \mathcal{GL} \ n, j \ \mathcal{GL} \ n, k \ \mathcal{GL} \ n)]_{o\sigma}]_{\sigma\sigma\sigma}$$

Where:

- E(x) stands for (Max n ≤ x)[Pr(n) Divides x] – Pr enumerates the prime numbers (e.g., Pr(0) stands for 2, Pr(5) stands for 13, etc.).

  x Divides y stands for the sentence $[\lambda x,y .\exists z .(z \leq y) \wedge (y = x \text{ Multiply } z)]$.

  x Multiply y stands for the function:

  $[\lambda x,y .@z .(y = 0 \wedge z = 0) \vee (z = (x \text{ Multiply } y-1) \text{ Plus } x)]$.[6]

  x Plus y stands for $[\lambda x,y .@z .(y = 0 \wedge z = x) \vee (z = (\text{Suc } x) \text{ Plus } y-1]$.

  Pr stands for the sentence:

  $[\lambda x. @z .(y = 0 \wedge z = 2) \vee (Pr(\text{Suc } z) = (\text{Min } m \leq \text{Suc}(Pr(z)!))[\text{Prime } m \wedge (Pr(z)<m)]$.

  Finally, Prime stands for $[\lambda x .(2 \leq x) \wedge (\forall y \leq x)[y \text{ Divides } x \rightarrow .y=x \vee y=1]$.

- n $\mathcal{GL}$ x stands for (Max y ≤ x)[$Pr(n)^y$ Divides x] – Here we use exponentiation.

  x^y stands for $[\lambda x,y .@z .(y = 0 \wedge z = 1) \vee (z = x \text{ Multiply } (x^{y-1}))]$.[7]

- Axiom(n) stands for [λn. ∃z. axiom(z,n)] – We devote an entire subsection to this function. For the present purpose, it is sufficient to note that this boolean term is true if and only if n represents a valid encoding of one the axioms.

- Rule(m,n,p) stands for [∃ exist wff's A, B, C, and D such that m = "[A = B]" and n = "C" and p = "D", and D is obtained from [A = B] and C by rule R]. Note rule R is the only rule of inference of Q_o and Q^-. As with Axiom, we devote an entire subsection to this function. For the present, Rule identifies whether the sentence m is derived from other sentences via application of a (the) rule of inference.

With this background, let us revisit the expression for proof. The first part of the expression contains $(m = E(n) \ \mathcal{GL} \ n)_o$ to ensure that the sentence encoded by m is indeed the last sentence in the series encoded by n.

The next part contains $(\forall k \leq E(n))[Axiom(k \ \mathcal{GL} \ n)]$. To understand this term, recall that Godelization of a series of sentences requires that the Godel encoding of each sentence be concatenated by using them as exponents to successively greater prime numbers. For example consider the sentence A. Its Godel encoding is "A". Now consider the sentence B. Its Godel encoding is "B". For the concatenation of the these sentences we would like AB to be encoded as "AB". We also want this encoding to be composable so that "A" * "B" = "AB" ≡ AB. Here's how: Passing from left to right, we associate each sentence with successive prime numbers. Then we use the Godel number of each sentence as the exponent to the prime number associated with each sentence. Thus, if "A" = 10000 and "B" = 100, then "AB" is $2^{10000} \times 3^{100}$.

---

[6] The definitions here for Multiply, Plus, and Pr differ from Andrew's Theory Q and from the HOL definition. We derived these definitions for their constructive nature and to demonstrate that using the Hilbert "e" operator can simplify the specification of a primitive recursive relationship.

The term ($\forall k \leq E(n)$) is important: it allows k to be bound from above by the index of the largest prime factor in the sequence. [Axiom(k $\mathcal{GL}$ n)] is then used to verify whether each sentence in the series is an axiom or derivation (result of applying a rule of inference). The factor in the sentence that "decides" whether some sentence in the series results from the application of a rule of inference is $[(\forall k \leq E(n) .(\exists i < k)(\exists j < k) .Rule(i\ \mathcal{GL}\ n, j\ \mathcal{GL}\ n, k\ \mathcal{GL}\ n)]$.

3.3. *Representing Axioms*

Given some Godel encoding of a sentence, we wish to determine if that sentence represents an axiom. An informal description of such a function is $[\exists z .\lambda n .(z=1 \wedge n =$ "Axiom 1") $\vee$ (z=2 $\wedge$ n = "Axiom 2") $\vee$ ... $\vee$ (z=W $\wedge$ n = "Axiom W")] which asserts the existence of z such that encoding represented by n is a valid axiom.

In the following subsections we define the expressions "Axiom x" in great detail.

3.3.1. *Axiom 1:* $g_{oo}T_o \wedge g_{oo}F_o = !x_o .g_{oo}x_o$

Axiom 1, which asserts that the only values representing the type truth are T and F, is very straight-forward: $g_{oo}T_o \wedge g_{oo}F_o = !x_o .g_{oo}x_o$.

The following shows the transformation of Axiom 1 from the form given to its more natural functional form:

$\wedge$ stands for $[\lambda h_{ooo} [Q_{ooo} [h_{ooo} T_o T_o]] [\lambda u_o \lambda v_o .h_{ooo} u_o v_o]]]$

Then, the lhs side becomes:

$[\lambda h_{ooo} .\lambda g_{oo} [[Q_{ooo} [h_{ooo} T_o T_o]] [\lambda u_o \lambda v_o .h_{ooo} u_o v_o]]] [g_{oo} T_o] [g_{oo} F_o]$

Unfortunately, Andrew's never defines function h. Indeed, it seems that function h cannot be defined in terms of equality alone.

Continuing with the right-hand side and after transforming the universal quantifier:

$[\lambda g_{oo} [Q_{ooo} [T_o]] [\lambda x_o .g_{oo} x_o]]]$

Since Andrew's did not provide a definition for function h and because "and" is easily expressed as a program:

if V(A) = T then if V(B) = T then return T else return F else return F

we have, after some rearrangement:

$[\lambda g_{oo} .\lambda x_o .[[Q_{ooo}[[\wedge_{ooo} [g_{oo} T_o]] [g_{oo} F_o]]][[Q_{ooo} [T_o]] [g_{oo} x_o]]]]$

Let axiom(1,n) stand for [λn .$\exists$g, x .g $\leq$ n $\wedge$ x $\leq$ n $\wedge$ n = "[λ" * $g^{\text{"o" * "o"}}$ * "[λ" * $x^{\text{"o"}}$ * "[[" * "Q"$^{\text{"o" * "o" * "o"}}$ * "[[" * "$\wedge$"$^{\text{"o" * "o" * "o"}}$ * "[[" * $g^{\text{"o" * "o"}}$ * "T"$^{\text{"o"}}$ * "]][" * $g^{\text{"o" * "o"}}$ * "F"$^{\text{"o"}}$ * "]]][[" * "Q"$^{\text{"o" * "o" * "o"}}$ * "[" * "T"$^{\text{"o"}}$ * "]][" * $g^{\text{"o" * "o"}}$ * $x^{\text{"o"}}$ * "]]]]"]

The definitions of * and "" are explained in the next subsection.

3.3.2. *Axiom 2:* $[X_\alpha = Y_\alpha] \supset .h_{o\alpha}X_\alpha = h_{o\alpha}Y_\alpha$

Axiom 2, on the other hand, is much more difficult because it, like most axioms, are schemata's where variables and types or both may be substituted (i.e., in Axiom 2, the type is polymorphic). For example, Axiom 2 is $[X_\alpha = Y_\alpha] \supset .h_{o\alpha}X_\alpha = h_{o\alpha}Y_\alpha$, but recall that = and $\supset$ are functions so that the actual arrangement of terms is $\supset_{ooo} Q_{o\alpha\alpha} X_\alpha Y_\alpha Q_{ooo} h_{o\alpha} X_\alpha h_{o\alpha} Y_\alpha$ and note that the type α is polymorphic over the expression.

---

[7] Exponentiation is somewhat complicated by the definition that $0^0 = 1$ but $0^{Suc\ y} = 0$.

If we properly bracket Axiom 2 based on function application and partial function application:

$$[[\supset_{ooo}\ [[Q_{o\alpha\alpha}\ X_\alpha\ ]\ Y_\alpha\ ]]\ [[Q_{ooo}\ [h_{o\alpha}\ X_\alpha\ ]]\ [h_{o\alpha}\ Y_\alpha]]]$$

The Godel encoding must consider the polymorphism of the type α and we also need to verify that the type substitution represents a valid type. Thus, we require a function to determine whether some non-zero number represents a valid type and this is not so simple as might be first thought. Types are Godelized as follows:

- Type o is Godelized, "o", as 3
- Type ι is Godelized, "ι", as 5
- Type $\alpha\beta$ is Godelized, "$\alpha\beta$", as $3^{\text{“}\alpha\text{”}} \times 5^{\text{“}\beta\text{”}}$

Thus, to Godelize a multi-symbol type, we traverse the series from left to right and compute $\Pi\ \Pr(i)^{\text{“symbol(i)”}}$ where i refers to the index position of each symbol and symbol(i) refers to the symbol at position i in the series. This Godelization process quickly demonstrates an important issue related to Godelization: The numbers quickly become very, very large.

Consider the type σ that we have used in previous sections to represent the type number as a shorthand for the type $o\iota$. It should not make a difference which we choose to use: We use σ to remind us of "number" and $o\iota$ to remind us that a number is just "that which is common" to all equivalent sets. The Godel number of the (simple) type $\sigma\sigma$ is difficult to use because it inherently means $((\sigma)(\sigma))$ which maps to $((o\iota)(o\iota))$. This means that "$\sigma\sigma$" is $3^{\text{“}\sigma\text{”}} \times 5^{\text{“}\sigma\text{”}}$ where "σ" is "$o\iota$" $\rightarrow 3^3 \times 5^3 \times 7^5$; now, this means that "$o\iota$" * "$o\iota$" = "$o\iota o\iota$" which is $3^3 \times 5^3 \times 7^5 \times 9^3 \times\ 11^3 \times 13^5$; so that "$o\iota$" = 56,723,625 and "$o\iota o\iota$" $\approx 2.0436 \times 10^{19}$ but, "σ" is $3^{56723625}$ and "$\sigma\sigma$" is $3^{56723625} \times 5^{56723625}$ = a really big number! Note also that "$((o\iota)(o\iota))$"is $3^{56723625} \times 5^{56723625}$.

Thus, determining whether a given number represents a type (i.e., some sequence of o and/or ι) or a type symbol (i.e., a symbol representing a type) is without too much complication.

If the number represents a type, the lambda expression

$$[\lambda n\ .\forall i\ .((i \leq E(n) \wedge (\Pr(i)^3 \text{ Divides } n \vee \Pr(i)^5 \text{ Divides } n)]$$

will be true. If the number represents a type symbol, the lambda expression

$$[\lambda m\ .\forall i\ .((i \leq E(m) \wedge \exists n\ .((\forall j\ .((j \leq E(n) \wedge (\Pr(j)^3 \text{ Divides } n \vee \Pr(j)^5 \text{ Divides } n)) \wedge \Pr(i)^n \text{ Divides } m))]$$

will be true. Now, letting Type stand for

$$[\lambda n\ .\forall i\ (i \leq E(n) \wedge (\Pr(i)^3 \text{ Divides } n \vee \Pr(i)^5 \text{ Divides } n))]$$

and TypeSymbol stand for

$$[\lambda m\ .\forall i\ .((i \leq E(m)) \wedge \exists n\ .((\forall j\ .((j \leq E(n) \wedge (\Pr(j)^3 \text{ Divides } n \vee \Pr(j)^5 \text{ Divides } n)) \wedge \Pr(i)^n \text{ Divides } m))]$$

We let ValidType[8] stand for

$$[\lambda n\ .\text{Type}(n) \vee \text{TypeSymbol}(n)]$$

[8] Our ValidType is a big improvement over Andrews [1]. In [1], his motivation is to show that these expressions were recursive. By building expressions that represent how a decision algorithm would operate, we demonstrate recursiveness in a manner more natural for the computer scientist.

and note that ValidType(n) = T_o if and only if, n represents a type or type symbol. With all of this, we are almost in a position to demonstrate the expression used to determine whether a given number represents a valid encoding of an instance of Axiom 2. We must first identify the Godel encoding of the "constant" symbols. The "constant" symbols of Axiom 2 (shown below) and their Godel encodings are computed below:

Axiom 2: $[[\supset_{ooo} [[Q_{o\alpha\alpha} X_\alpha] Y_\alpha]] [[Q_{ooo} [h_{o\alpha} X_\alpha]] [h_{o\alpha} Y_\alpha]]]$

Where,

$\text{“}h_{o\alpha}\text{”} = 31^{\text{“}o\alpha\text{”}} = 31\exp(3^3 \times 5^{\text{“}\alpha\text{”}})$

$\text{“}X_\alpha\text{”} = 103^{\text{“}\alpha\text{”}} = 103\exp(3^{\text{“}\alpha\text{”}})$

$\text{“}Y_\alpha\text{”} = 107^{\text{“}\alpha\text{”}} = 107\exp(3^{\text{“}\alpha\text{”}})$

$\text{“}Q_{o\alpha\alpha}\text{”} = 19^{\text{“}o\alpha\alpha\text{”}} = 19\exp(3^3 \times 5^{\text{“}\alpha\text{”}} \times 7^{\text{“}\alpha\text{”}})$

Finally[9],

Let axiom(2,n) stand for $[\lambda n .\exists a .a \le n .\text{ValidType}_o(a) \wedge n = \text{“}[[\supset_{ooo}[[\text{”} * 19^{(3^3 \times 5^{\text{“}a\text{”}} \times 7^{\text{“}a\text{”}})} * 103^{\text{“}a\text{”}} * \text{“}]\text{”} * 107^{\text{“}a\text{”}} * \text{“}]][[\text{”} * 19^{(3^3 \times 5^3 \times 7^3)} * 31^{(3^3 \times 5^{\text{“}a\text{”}})} * 103^{\text{“}a\text{”}} * \text{“}]][\text{”} * 31^{(3^3 \times 5^{\text{“}a\text{”}})} * 107^{\text{“}a\text{”}} * \text{“}]]]\text{”}]$

3.3.3. Axiom 3: $f_{\alpha\beta} = g_{\alpha\beta} = !X_\beta .f_{\alpha\beta}X_\beta = g_{\alpha\beta}X_\beta$

Axiom 3, $f_{\alpha\beta} = g_{\alpha\beta} = !X_\beta .f_{\alpha\beta}X_\beta = g_{\alpha\beta}X_\beta$, when rewritten in functional form becomes:

$[[Q_{ooo}[[Q_{o(\alpha\beta)(\alpha\beta)}F_{(\alpha\beta)}]G_{(\alpha\beta)}]][[Q_{o(o\beta)(o\beta)}[\lambda x_\beta T_o]]][\lambda x_\beta[[Q_{o\alpha\alpha}[F_{(\alpha\beta)}x_\beta]][G_{(\alpha\beta)}x_\beta]]]]$

and its Godelization:

Let axiom(3,n) stand for $[\lambda n .\exists \alpha,\beta .\alpha \le n \wedge \beta \le n .\text{ValidType}_o(\alpha) \wedge \text{ValidType}_o(\beta) \wedge n = \text{“}[[\text{”} * \text{“}Q\text{”}^{\text{“}o\text{”} * \text{“}o\text{”} * \text{“}o\text{”}} * \text{“}[[\text{”} * \text{“}Q\text{”}^{\text{“}o\text{”} * \text{“}(\alpha\beta)\text{”} * (3^{\text{“}\alpha\text{”}} \times 5^{\text{“}\beta\text{”}})} * \text{“}F\text{”}^{\text{“}(\alpha\beta)\text{”}} * \text{“}]\text{”} * \text{“}G\text{”}^{\text{“}(\alpha\beta)\text{”}} * \text{“}]][[\text{”} * \text{“}Q\text{”}^{\text{“}o\text{”} * \text{“}(o\beta)\text{”} * (3^{\text{“}o\text{”}} \times 5^{\text{“}\beta\text{”}})} * \text{“}[\lambda\text{”} * \text{“}x\text{”}^{\text{“}\beta\text{”}} * \text{“}T\text{”}^{\text{“}o\text{”}} * \text{“}]][\lambda\text{”} * \text{“}x\text{”}^{\text{“}\beta\text{”}} * \text{“}[[Q\text{”}^{\text{“}o\text{”} * \text{“}\alpha\text{”} * \text{“}\alpha\text{”}} * \text{“}[\text{”} * \text{“}F\text{”}^{\text{“}(\alpha\beta)\text{”}} * \text{“}x\text{”}^{\text{“}\beta\text{”}} * \text{“}]][\text{”} * \text{“}G\text{”}^{\text{“}(\alpha\beta)\text{”}} * \text{“}x\text{”}^{\text{“}\beta\text{”}} * \text{“}]]]]]\text{”}]$

Note that in axiom(3,n) we have made the following simplifications:

- We use "Q", "x", etc., to represent the Godel number/encoding for the object being quoted;
- λx introduces an abstraction – x represents any valid variable name and axiom(3,n) should have the conjunctive term $(\exists x \le n .\text{ValidVariable}(x))$ and everywhere we have used "x" we should have used x (the Godel number associated with the valid variable name); and,
- Except for one instance, we have favored the simpler form "(αβ)" for types over the more complex (and less obvious) $(3^{\text{“}\alpha\text{”}} \times 5^{\text{“}\beta\text{”}})$.

For completeness, let ValidVariable(n) stand for [λn .∃a, i .a ≤ n ∧ i > 'number of primitive symbols' ∧ ValidType(a) ∧ n = Pr(i)^a].

3.3.4. Axiom 4: $[\lambda x_\alpha B_\beta]A_\alpha = B_\beta$

Axiom 4, $[\lambda x_\alpha B_\beta]A_\alpha = B_\beta$, rewritten in functional form: $[[Q_{o\beta\beta}[[\lambda x_\alpha B_\beta] A_\alpha]] B_\beta]$

---

[9] Our axiom 2 corrects a number of serious errors in Andrews. His encoding of Axiom 2 is not composable, as is ours.

Let axiom(4,n) stand for [λn .∃x, B, A, α, β .α ≤ n ∧ β ≤ n ∧ A ≤ n ∧ B ≤ n ∧ x ≤ n ∧ ValidVariable(x) ∧ WFF(A) ∧ TypeOfWFF(A) = TypeOfVariable(x) ∧ ValidSymbol(B) ∧ B ≠ x ∧ ValidType(α) ∧ ValidType(β) ∧ n = "[[" * "Q"$^{\text{“o” * “β” * “β”}}$ * "[[λ" * x$^{\text{“α”}}$ * B$^{\text{“β”}}$ * "]" * A$^{\text{“α”}}$ * "]]" * B$^{\text{“β”}}$ * "]"]

Where,

- ValidSymbol(n) stands for [λn .ValidVariable(n) ∨ n = Pr(index of primitive symbol ι)$^{\text{“ι(oι)”}}$ ∨ (.∃a .a ≤ n ∧ ValidType(a) ∧ n = "Q"$^{\text{“o” * “a” * “a”}}$)]
- TypeOfWFF(n) stands for [λn .@y .min y ≤ n .∃z ≤ n .n = z^y ∧ WFF(z)]
- TypeOfVariable(n) stands for [λn .@y .min y ≤ n .∃z ≤ n .n = z^y ∧ Prime(z)]

3.3.4.1. Parsing as a Type-Theoretic Expression

The function $WFF_{\sigma\to o}$ is a very complicated expression. In what follows we demonstrate that WFF is a computable function of numbers to boolean. We then show how one would express this function as a type-theoretic expression.

Consider the following type-theoretic expression[10]:

$$[\exists n_{o\iota}, m_{o\iota} .\forall x_\iota [m_{o\iota}\, x_\iota \to n_{o\iota}\, x_\iota] \wedge \exists y_\iota [n_{o\iota}\, y_\iota \wedge \neg m_{o\iota}\, y_\iota] \wedge E_{o(o\iota)(o\iota)}\, n_{o\iota}\, m_{o\iota}]$$

It asserts the existence of two sets (m and n) such that for all individuals, x, if x is a member of m, then x is a member of n (m is a subset of n). The expression continues by asserting the existence of an individual y that is in n, but not in m; thus, m is a proper subset of n. The last term says that the two sets can be brought into one-to-one correspondence. Now that we are convinced that the sentence makes sense, how do we "prove" that it is a wff?.

In order for a sentence to be well-formed, it must be syntactically correct and for a type-theoretic system, types are part of the syntax. Thus, in the above expression, we can state that if all of the sub-terms are wff, then the sentence is wff. For example, consider the sub-term involving the existential y as it more properly written: $[\wedge_{ooo}\, [[n_{o\iota}\, y_\iota]]\, [\neg m_{o\iota}\, y_\iota]]$. This sub-term is wff, since each of the factors resolve to the proper type[11].

For type theory, we can build a parser to verify that a sentence is a wff. Here's how:

- Simplest approach: Do not try to parse the number theoretic sentence. As part of the Godelization process, do the parse; thus, if and only if the sentence is wff will our Godelizer generate a number.
- More difficult: Although complex, we can extract the unique sentence represented by the Godel number being considered. This sentence is then passed to the parser which returns success if the sentence is wff and fails otherwise.

If well-formedness is essentially the parsing problem, then we can imagine a Turing machine "programmed" with our parser. Cook's Theorem tells us that we can translate the parser to an array of machine descriptions, such that the first row is the instantaneous description of the initial machine configuration with the input sentence. Each subsequent row is a legal successor to the previous row. The last row contains either accept or reject.

[10] This expression is another representation of infinity. It says that a set is infinite if and only if it can be put into one-to-one correspondence with a proper subset of itself.

[11] HOL allows us to write these functions, including the parser for wff in ML. If we were not able to do this, we would be in the tedious position of having to actually implement a propositional logic formulation of the parser – we discuss this application of Cook's Theorem in subsequent paragraphs.

Finally, we define propositional variables (using only propositional logic) to "simulate" this array. Thus, variable q(t, i) is true if our Turing machine simulated tape contains variable q on square i at time t. Since each square (we know in advance that the upper bound of the number of squares need to parse a sentence is related to the size of the sentence) contains at least symbol at all times, then we have $q(t, i) \vee a(t, i) \vee \ldots \vee Q(t, i)$. Since each square can only have one symbol at each time, we also have $\sim(q(t, i) \vee a(t, i))$ for each pair of symbols.

At this point the reader should be convinced that we can build a propositional logic expression that describes the operation of the parser in such a manner that we can determine whether the parser accepted or rejected the sentence. Actually, the situation is much more profound: We can build such an expression for any recursive set including our metatheory. Godel's Incompleteness Theorem teaches us, though, that when presented with sentence G, that the system will not be able to decide on its status as a theorem.

3.3.5. Axiom 5: $[\lambda x_\alpha x_\alpha]A_\alpha = A_\beta$

Axiom 5, $[\lambda x_\alpha x_\alpha]A_\alpha = A_\beta$, in functional form: $[[Q_{o\alpha\alpha}[[\lambda x_\alpha x_\alpha]A_\alpha]]\ A_\alpha]$

Let axiom(5,n) stand for [λn .∃A, x, α .α ≤ n ∧ x ≤ n ∧ A ≤ n ∧ ValidVariable(x) ∧ ValidType(α) ∧ WFF(A) ∧ TypeOf(A) = PrimeLog(x) ∧ n = "[[" * "Q"$^{\text{“o” * “α” * “α”}}$ * "[[λ" * x$^{\text{“α”}}$ * x$^{\text{“α”}}$ * "]" * A$^{\text{“α”}}$ * "]]" * A$^{\text{“α”}}$ * "]"]

3.3.6. Axiom 6: $[\lambda x_\alpha .B_{\beta\gamma}C_\gamma]A_\alpha = [[\lambda x_\alpha B_{\beta\gamma}]A_\alpha]\ [[\lambda x_\alpha C_\gamma]A_\alpha]$

Axiom 6, $[\lambda x_\alpha .B_{\beta\gamma}C_\gamma]A_\alpha = [[\lambda x_\alpha B_{\beta\gamma}]A_\alpha]\ [[\lambda x_\alpha C_\gamma]A_\alpha]$, in functional form is quite daunting:

$$[[Q_{o\beta\beta}[[\lambda x_\alpha[B_{\beta\gamma}C_\gamma]]A_\alpha]][[[\lambda x_\alpha B_{\beta\gamma}]A_\alpha][[\lambda x_\alpha C_\gamma]A_\alpha]]]$$

Let axiom(6,n) stand for [λn .∃A, B, C, x, α, β, γ .α ≤ n ∧ β ≤ n ∧ γ ≤ n ∧ x ≤ n ∧ A ≤ n ∧ B ≤ n ∧ C ≤ n ∧ ValidVariable(x) ∧ ValidType(α) ∧ ValidType(β) ∧ ValidType(γ) ∧ WFF(A) ∧ WFF(B) ∧ WFF(C) ∧ TypeOf(A) = PrimeLog(x) ∧ TypeCompatible(TypeOf(B),TypeOf(C)) ∧ n = "[[" * "Q"$^{\text{“o” * “β” * “β”}}$ * "[[λ" * x$^{\text{“α”}}$ * "[" * B$^{\text{“βγ”}}$ * C$^{\text{“γ”}}$ * "]]" * A$^{\text{“α”}}$ * "]][[[λ" * x$^{\text{“α”}}$ * B$^{\text{“βγ”}}$ *"]" * A$^{\text{“α”}}$ * "][[λ" * x$^{\text{“α”}}$ * C$^{\text{“γ”}}$ *"]" * A$^{\text{“α”}}$ * "]]]"]

TypeCompatibilty anticipates extensions of well-formedness to include parsing the entire sentence so that we may claim that if a sentence is well-formed then so are its sub-terms.

3.3.7. Axiom 7: $[\lambda x_\alpha .\lambda y_\gamma B_\delta]A_\alpha = [\lambda y_\gamma[\lambda x_\alpha B_\delta]A_\alpha]$

The functional form for Axiom 7: $[[Q_{o(\delta\gamma)\delta\gamma}[[\lambda x_\alpha[\lambda y_\gamma B_\delta]]A_\alpha]][\lambda y_\gamma[[\lambda x_\alpha B_\delta]A_\alpha]]]$

and let axiom(7,n) stand for [λn .∃A, B, x, y, α, δ, γ .α ≤ n ∧ δ ≤ n ∧ γ ≤ n ∧ x ≤ n ∧ y ≤ n ∧ A ≤ n ∧ B ≤ n ∧ ValidVariable(x) ∧ ValidVariable(y) ∧ y ≠x ∧ValidType(α) ∧ ValidType(δ) ∧ ValidType(γ) ∧ WFF(A) ∧ WFF(B) ∧ TypeOf(A) = PrimeLog(x) ∧ n = "[[" * "Q"$^{\text{“o” * “(δγ)” * (3}^{\text{“δ”}} \times 5^{\text{“γ”}})}$ * "[[λ" * x$^{\text{“α”}}$ * "[λ" * y$^{\text{“γ”}}$ * B$^{\text{“δ”}}$ * "]]" * A$^{\text{“α”}}$ * "]][λ" * y$^{\text{“γ”}}$ * "[[λ" * x$^{\text{“α”}}$ * B$^{\text{“δ”}}$ * "]" * A$^{\text{“α”}}$ * "]]]"]

3.3.8. Axiom 8: $[\lambda x_\alpha .\lambda x_\alpha B_\delta]A_\alpha = [\lambda x_\alpha B_\delta]$

In functional form, Axiom 8: $[[Q_{o(\delta\alpha)(\delta\alpha)}[[\lambda x_\alpha B_\delta]A_\alpha]][\lambda x_\alpha B_\delta]]$

Let axiom(8,n) stand for [λn .∃A, B, x, α, δ .α ≤ n ∧ δ ≤ n ∧ x ≤ n ∧ A ≤ n ∧ B ≤ n ∧ ValidVariable(x) ∧ ValidType(α) ∧ ValidType(δ) ∧ WFF(A) ∧ WFF(B) ∧ TypeOf(A) = PrimeLog(x) ∧ n = "[[" * "Q"$^{\text{“o” * “(δα)” * (3}^{\text{“δ”}} \times 5^{\text{“α”}})}$ * "[[λ" * x$^{\text{“α”}}$ * B$^{\text{“δ”}}$ * "]" * A$^{\text{“α”}}$ * "]][λ" * x$^{\text{“α”}}$ * B$^{\text{“δ”}}$ * "]]"]

3.3.9. *Axiom 9:* $[\iota_{\iota(o\upsilon)}[Q_{o\iota\iota}\, y_\iota]=y_\iota]$

The functional representation of Axiom 9 is similar to axiom 2: $[[Q_{o\iota\iota}[y_\iota]][\iota_{\iota(o\upsilon)}[Q_{o\iota\iota}[y_\iota]]]]$.

Let axiom(9,n) stand for [λn .∃y .y ≤ n ∧ ValidVariable(y) ∧ n = "[[" * "Q"$^{\text{“o” * “o” * “ou”}}$ * "[" * y$^{\text{“ι”}}$ * "]][" * "ι"$^{\text{“o” * “ou”}}$ * "[" * "]][" * "Q"$^{\text{“o” * “o” * “ou”}}$ * "[" * y$^{\text{“ι”}}$ * "]]]]"]

3.4. *Representing Rules of Inference*

Q_0 has only one rule of inference: From C and A_α=B_α to infer the result of replacing one occurrence of A_α in C by an occurrence of B_α, provided that the occurrence of A_α in C is not an occurrence of a variable immediately preceded by λ.

This means that there is some m such that m = "[[$Q_{o\alpha\alpha}$ A_α] B_α]" and formulas (not necessarily well-formed) J and K (possibly empty), such that C = "J" * A_α * "K" and D = "J" * B_α * "K".

Let Rule(m,n,p) stand for [λm .λn .λp .∃A, B, α, z .A ≤ m ∧ B ≤ m ∧ α ≤ m ∧ wff(A) ∧ wff(B) ∧ TypeOf(A) = TypeOf(B) ∧ TypeOf(A) = z ∧ m = "[[" * "Q"$^{\text{“o” * z * z}}$ * A^z * "]" * B^z * "]" ∧ TypeOf(n) = "o" ∧ TypeOf(p) = "o" ∧ .∃j, k .j≤ n ∧ k ≤ n ∧ n = j * A^z * k ∧ p = j * B^z * k]

4. Demonstrating GIT in HOL

Section 3, "Godel's Incompleteness Theorem" expressed a backwards proof of GIT in HOL. In this section, we are concerned with the implementation details of using the HOL theorem prover to demonstrate that we have a correct proof.

Godel's sentence G, $[\lambda x_\sigma\ \forall y_\sigma \sim Proof_{\sigma\sigma\sigma}{}^{S}\ [\ "[" * x_\sigma * [Num_{\sigma\sigma}\ x_\sigma] * "]"\ ]\ y_\sigma]_{\sigma\to\sigma}$, demands that there be no integer y that encodes a Proof of "G" G. Since all HOL proofs are constructive, we must show that any such Proof of "G" G leads to contradiction.

An interesting approach to the problem would be to show that Godel's Incompleteness Theorem is reducible to another problem. For example, consider the following program.

```
Program P:
      For i = 0 to ∞
      do
            if i encodes a proof that P does not halt, then halt
      od;
End P.
```

By construction: P halts iff there is a proof that P does not halt.

Let $\psi \equiv$ "P does not halt"

Then,

$$\psi \to \sim \text{Proof}(\sim \text{Halt P}) \equiv \psi \to \sim \text{Proof}(\psi)$$

and

$$\sim\psi \to \text{Proof}(\sim \text{Halt P}) \equiv \sim\psi \to \text{Proof}(\psi)$$

This program fragment shows a reducibility of Godel's sentence G to the halting problem. The proof relies on a violation of ω-consistency, $\vdash \exists\omega F(\omega) \to \exists\omega F(\omega)$, as in Godel's original proof.

Whether we choose Godel's original sentence G or a reduction of it, we shall always be faced with the impossibility of showing constructive proof. The problem, of course, is that we need to show that each integer does not encode a valid proof. Even though the cardinality of this set is "countably" infinite, it is still infinite none-the-less. Unfortunately, incompleteness means that within the formal system, Godel's sentence G is undecidable – so that neither Proof "G" G nor ~Proof "G" G can be determined within the formal system.

Thus, given a correctly implemented proof, the result cannot be decided constructively. To implement proof-by-contradiction in HOL, requires a different strategy. One first expresses an implication (e.g., $p \to q$) whose negation is provable: $\sim(p \to q) \equiv \sim(T \to F) \equiv \sim F$.

The following example shows how, at a high-level, Godel's Incompleteness Theorem would be demonstrated using the HOL theorem prover.

Every backward proof begins with a goal. In HOL, a goal is specified using the set_goal function. This function has two arguments: 1) A list of terms corresponding to the assumptions of the goal; and 2) a term specifying the goal.

set_goal([],string_to_term "?G. ~(!g:num . (?x:num . G g x) ==> (!y . ~ G g y))");

HOL places the goal on the goal stack:

```
val it =
 Status: 1 proof.
 1. Incomplete:
```

```
Initial goal:
(--'?G. ~(!g. (?x. G g x) ==> (!y. ~(G g y)))'--)
: proofs
```

For illustrative purposes we instantiate G using the "EXISTS_TAC". Note that function e is an abbreviation for expand, a function that applies tactics (and tacticals) to the goal stack.

- e(EXISTS_TAC(string_to_term "\\(x:num) (y:num). T"));

The "EXISTS_TAC" replaces every occurrence of G with the specified lambda-expression. HOL responds with:

```
OK..
1 subgoal:
val it =
 (--'~(!g. (?x. (\x y. T) g x) ==> (!y. ~((\x y. T) g y)))'--)
 : goalstack
```

We then facilitate a β-conversion reduction using "BETA_TAC":

- e(BETA_TAC);

HOL responds by "doing" the conversion and displaying the result:

```
OK..
1 subgoal:
val it =
 (--'~(!g. (?x. T) ==> (!y. ~T))'--)
 : goalstack
```

Next we convert the subgoal to prenex format using the PRENEX_CONV conversion. In HOL conversions rewrite a term and then assert a theorem stating that the original term implies the rewritten term. The CONV_TAC tactic puts the original term in the assumption (undischarge) so that the rewritten term is the new subgoal.

- e(CONV_TAC PRENEX_CONV);

The result shows that all quantifiers have been moved to the extreme left-hand side of the equation:

```
OK..
1 subgoal:
val it =
 (--'?g y x. ~(T ==> ~T)'--)
 : goalstack
```

Next the existentials are removed. REPEAT is used to cause EXISTS_TAC to be applied as many times as possible.

- e(REPEAT (EXISTS_TAC (string_to_term "1")));

Since all quantifiers were moved to the extreme left (prenex form) and all of them were existential, they are all removed by repeatedly applying "EXIST_TAC":

```
OK..
1 subgoal:
val it =
 (--'~(T ==> ~T)'--)
 : goalstack
```

Finally, the tautology is eliminated:

- e(CONV_TAC Taut.TAUT_CONV);

The HOL theorem prover recognizes that the goal is proved, informs the user, and then displays the proof:

```
OK..
Goal proved.  |- ~(T ==> ~T)

Goal proved.  |- ?g y x. ~(T ==> ~T)

Goal proved.  |- ~(!g. (?x. T) ==> (!y. ~T))

Goal proved.  |- ~(!g. (?x. (\x y. T) g x) ==> (!y. ~((\x y. T) g y)))
val it =
  Initial goal proved.
  |- ?G. ~(!g. (?x. G g x) ==> (!y. ~(G g y))) : goalstack
```

To actually verify GIT, one needs to instantiate G with Godel's sentence and then continue with definitions for Proof, etc., as we did in section 3.

5. Conclusion

5.1. *Summary*

Even with the passage of nearly four decades, Godel's Incompleteness Theorem remains difficult. Other researchers have reworked and reinterpreted the theorem, but the details are still formidable and quite challenging. Godel has shown that any recursively axiomatizable formal system (i.e., any formal system whose axioms and rules of inference are computable) is incomplete or inconsistent. The ramifications run deep and wide: Can a rational system capture it's own metatheory?. If not, is the irrationality of man simply a reflection of Godel's Incompleteness Theorem? Does the mimicry of intelligence stand for intelligence or do we remain forever undecided?

In this work we have taken the novel and original approach of presenting Godel's First Incompleteness Theorem in reverse. Even more fundamental is that we show how to compute the sub-goals. This is a radical departure from every other presentation of Godel's Incompleteness Theorem – traditionally, the terms, predicates, etc., comprising Godel's Theorem are shown to be general recursive.

The Church–Turing thesis states that our notion of computability is expressed equally and completely by general recursion or Turing machines; therefore, Godel's Incompleteness Theorem may be expressed equally and completely within the context of a program instead of a set of general recursive functions.

In this regard, HOL is an ideal theorem prover. The HOL theorem prover relies on the programming language, ML (Meta-Language) as the meta-language for HOL. ML was developed as part of the LCF (Logic for Computable Functions) system for interactive automated reasoning about higher-order, recursively defined functions. Thus, HOL and ML are ideal for expressing metamathematics and related metatheory. ML is especially useful as a bridge between general recursion and Von-Neumann style imperative programming paradigms.

5.2. *Problems Encountered*

Godel's Incompleteness Theorem is very complex. A number of authors [16] have attempted to make Godel's work accessible to the non-mathematician. Unfortunately, these attempts generally fail to capture important notions such as proof, validity, consistency, soundness, and completeness. Other authors [1,3,7,12,17,18], writing for mathematicians (and/or computer scientists), are quite liberal in their approach and notation; thus, making each of them incomparable. In the end, one is forced to go through the original proof and through each of the various authors; eventually, of course, one understands.

Type theory is relatively simple, but has a moderate learning curve. HOL, on the other hand, is quite complex and has a steep learning curve. The HOL documentation is well-written, but for a system called HOL-88, not HOL-90 which was used in this work. Unhappily, the differences between HOL-90 and HOL-88 syntax are not always easily determined. Overtime, as one works through the dual learning curves of HOL and the syntax differences, the HOL system becomes less difficult.

One area of difficulty that was not actually encountered (but still very important) involves the use of numbers. In Godel's Theorem, we "Godelize" expressions, equations, etc., in order to demonstrate the concept of proof, etc., numbers. As we have discussed, our approach is not number theoretic but program-theoretic; thus, we can avoid the problems of big numbers by handling "Godelization" symbolically.

5.3. *Suggestions for Further Work*

Godel's First Incompleteness Theorem when expressed solely in terms of recursive function theory is extremely complex. A type-theoretic presentation of the theorem is simpler to understand; however, the relationship between type theory and general recursion is not obvious. Due to time limitations, etc., it was not possible to completely encode Godel's First Incompleteness Theorem with the HOL theorem prover.

A useful and worthy project would be to complete the encoding and to build a parser for symbolic Godelization.

6. References

[1] P.B. Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[2] H.P. Barendregt, *The Lambda Calculus Its Syntax and Semantics*, North-Holland, 1984.

[3] G. Boolos, *The Logic of Provability*, Cambridge University Press, 1993.

[4] R. Carnap, *Introduction to Symbolic Logic*, Dover, 1958.

[5] A. Church, "A formulation of the simple theory of types," J. Symbolic Logic, Vol. 5, pp. 56-68, 1940.

[6] A. Church, *Introduction to Mathematical Logic*, Princeton University Press, 1996.

[7] M. Davis, *Computability and Unsolvability*, Dover, 1973.

[8] A. DiCanzio, *Galileo*, ADASI, 1996

[9] K. Godel, *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, Dover, 1992.

[10] M. Gordon, "HOL – A Machine Oriented Formulation of Higher Order Logic," University of Cambridge Computer Laboratory Technical Report No. 68, July 1985.

[11] M.J.C. Gordon and T.F. Melham, *Introduction to HOL*. Cambridge University Press, 1993.

[12] G. Hunter, *Metalogic An Introduction to the Metatheory of Standard First Order Logic*, University of California Press, 1996.

[13] R.M. Karp, "Lecture Notes on Reducibility and NP-Completeness," UC Berkeley.

[14] L.C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1991.

[15] S. MacLane, *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[16] E. Nagel and J.R. Newman, *Godel's Proof*, New York University Press, 1986.

[17] N. Shankar, *Metamathematics, Machines, and Godel's Proof*. Cambridge University Press, 1994.

[18] R.M. Smullyan, *Godel's Incompleteness Theorems*. Oxford University Press, 1992.

[19] A. Tarski, *Introduction to Logic and to the Methodology of Deductive Sciences*, Dover, 1995.

[20] R. Toal, "Toward Automated Compiler Verification," Ph.D. Dissertation, UCLA, 1993.

[21] J. von Wright, "Representing higher-order logic proofs in HOL," The Computer Journal, Vol. 38, No. 2, pp. 171-179, July 1995.