A Mechanised Proof of Gödel's Incompleteness Theorems using Nominal Isabelle

Lawrence C. Paulson

Abstract An Isabelle/HOL formalisation of Gödel's two incompleteness theorems is presented. The work follows Świerczkowski's detailed proof of the theorems using hereditarily finite (HF) set theory [32]. Avoiding the usual arithmetical encodings of syntax eliminates the necessity to formalise elementary number theory within an embedded logical calculus. The Isabelle formalisation uses two separate treatments of variable binding: the nominal package [34] is shown to scale to a development of this complexity, while de Bruijn indices [3] turn out to be ideal for coding syntax. Critical details of the Isabelle proof are described, in particular gaps and errors found in the literature.

1 Introduction

This paper describes mechanised proofs of Gödel's incompleteness theorems [8], including the first mechanised proof of the second incompleteness theorem. Very informally, these results can be stated as follows:

Theorem 1 (First Incompleteness Theorem) *If L is a consistent theory capable of formalising a sufficient amount of elementary mathematics, then there is a sentence δ such that neither δ nor $\neg\delta$ is a theorem of L, and moreover, δ is true.*[1]

Theorem 2 (Second Incompleteness Theorem) *If L is as above and* $\mathrm{Con}(L)$ *is a sentence stating that L is consistent, then* $\mathrm{Con}(L)$ *is not a theorem of L.*

Both of these will be presented formally below. Let us start to examine what these theorems actually assert. They concern a consistent formal system, say L, based on first-order logic with some additional axioms: Gödel chose Peano arithmetic (PA) [7], but hereditarily finite (HF) set theory is an alternative [32], used here. The first theorem states that any such axiomatic system must be incomplete, in the sense that some sentence can neither be proved nor disproved. The expedient of adding that sentence as an axiom merely creates a new axiomatic system, for which there is another undecidable sentence. The theorem can be strengthened to allow infinitely many additional axioms,

Computer Laboratory, University of Cambridge, England
E-mail: lp15@cl.cam.ac.uk

[1] Meaning, δ (which has no free variables) is true in the standard model for L.

provided there is an effective procedure to recognise whether a given formula is an axiom or not.

The second incompleteness theorem asserts that the consistency of L cannot be proved in L itself. Even to state this theorem rigorously requires first defining the concept of provability in L; the necessary series of definitions amounts to a computer program that occupies many pages. Although the same definitions are used to prove the first incompleteness theorem, they are at least not needed to state that theorem. The original rationale for this project was a logician's suggestion that the second incompleteness theorem had never been proved rigorously. Having completed this project, I sympathise with his view; most published proofs contain substantial gaps and use cryptic notation.

Both incompleteness theorems are widely misinterpreted, both in popular culture and even by some mathematicians. The first incompleteness theorem is often taken to imply that mathematics cannot be formalised, when evidently it has been, this paper being one of numerous instances. It has also been used to assert that human intelligence can perceive truths (in particular, the truth of δ, the undecidable sentence) that no computer will ever understand. Franzén [5] surveys and demolishes many of these fallacies. The second incompleteness theorem destroyed Hilbert's hope that the consistency of quite strong theories might be proved even in Peano arithmetic. It also tells us, for example, that the axioms of set theory do not imply the existence of an inaccessible cardinal, as that would yield a model for set theory itself.

The first incompleteness theorem has been proved with machine assistance at least three times before. The first time (surprisingly early: 1986) was by Shankar [29, 30], using Nqthm. Then in 2004, O'Connor [22] (using Coq) and Harrison (using HOL Light)[2] each proved versions of the theorem. The present proof, conducted using Isabelle/HOL, is novel in adopting nominal syntax [34] for formalising variable binding in the syntax of L, while using de Bruijn notation [3] for coding those formulas. Despite using two different treatments of variable binding, the necessary representation theorem for formulas is not difficult to prove. It is not clear that other treatments of higher-order abstract syntax could make this claim. These proofs can be seen as an extended demonstration of the power of nominal syntax, while at the same time vindicating de Bruijn indexing in some situations.

The machine proofs are fairly concise at under 12,400 lines for both theorems.[3] The paper presents highlights of the proof, commenting on the advantages and disadvantages of the nominal framework and HF set theory. An overview of the project from a logician's perspective has appeared elsewhere [27]. The proof reported here closely follows a detailed exposition by Świerczkowski [32]. His careful and detailed proofs were indispensable, despite some errors and omissions, which are reported below. For the first time, we have complete, formal proofs of both theorems. They take the form of structured Isar proof scripts [26] that can be examined interactively.

The remainder of the paper presents background material (Sect. 2) before outlining the development itself: the proof calculus (Sect. 3), the coding of the calculus within itself (Sect. 4) and finally the first theorem (Sect. 5). Technical material relating to the second theorem are developed (Sect. 6) then the theorem is presented and discussed (Sect. 7). Finally, the paper concludes (Sect. 8).

[2] Proof files at `http://code.google.com/p/hol-light/`, directory `Arithmetic`

[3] This is approximately as long as Isabelle's theory of Kurzweil-Henstock gauge integration.

2 Background

Isabelle/HOL [20] is an interactive theorem prover for higher-order logic. This formalism can be seen as extending a polymorphic typed first-order logic with a functional language in which recursive datatypes and functions can be defined. Extensive documentation is available.[4]

For interpreting the theorem statements presented below, it is important to note that a theorem that concludes ψ from the premises $\phi_1, \ldots, \phi_n$ may be expressed in a variety of equivalent forms. The following denote precisely the same theorem:

$[\![\phi_1; \ \ldots; \ \phi_n]\!] \Longrightarrow \psi$
$\phi_1 \Longrightarrow \cdots \Longrightarrow \phi_n \Longrightarrow \psi$
assumes ϕ_1 **and** $\ldots$ **and** ϕ_n **shows** ψ

If the conclusion of a theorem involves the keyword **obtains**, then there is an implicit existential quantification. The following two theorems are logically equivalent.

$\phi \Longrightarrow \exists x.\ \psi_1 \wedge \ldots \wedge \psi_n$
assumes ϕ **obtains** x **where** $\psi_1 \ \ldots \ \psi_n$

Other background material for this paper includes an outline of Gödel's proof, an introduction to hereditarily finite set theory and some examples of Nominal Isabelle.

2.1 Gödel's Proof

Much of Gödel's proof may be known to many readers, but it will be useful to list the milestones here, for reference.

1. A first-order *deductive calculus* is formalised, including the syntax of terms and formulas, substitution, and semantics (evaluation). The calculus includes axioms to define Peano arithmetic or some alternative, such as the HF set theory used here. There are inference rules for propositional and quantifier reasoning. We write $H \vdash A$ to mean that A can be proved from H (a set of formulas) in the calculus.
2. Meta-theory is developed to relate *truth and provability*. The need for tedious proof constructions in the deductive calculus is minimised through a meta-theorem stating that a class of true formulas are theorems of that calculus. One way to do this is through the notion of Σ formulas, which are built up from atomic formulas using $\vee$, $\wedge$, $\exists$ and bounded universal quantification. Then the key result is

$$\text{If } \alpha \text{ is a true } \Sigma \text{ sentence, then } \vdash \alpha. \tag{1}$$

3. A *system of coding* is set up within the formalised first-order theory. The code of a formula α is written $\ulcorner\alpha\urcorner$ and is a term of the calculus itself.
4. In order to *formalise the calculus within itself*, predicates to recognise codes are defined, including terms and formulas, and syntactic operations on them. Next, predicates are defined to recognise individual axioms and inference rules, then a sequence of such logical steps. We obtain a predicate Pf , where $\vdash \mathrm{Pf}\ \ulcorner\alpha\urcorner$ expresses that the formula α has a proof. The key result is

$$\vdash \alpha \iff \vdash \mathrm{Pf}\ \ulcorner\alpha\urcorner. \tag{2}$$

[4] `http://isabelle.in.tum.de/documentation.html`

All of these developments must be completed before the second incompleteness theorem can even be stated.

5. Gödel's *first* incompleteness theorem is obtained by constructing a formula δ that is provably equivalent (within the calculus) to the formal statement that δ is not provable. It follows (provided the calculus is consistent) that neither δ nor its negation can be proved, although δ is true in the semantics.
6. Gödel's *second* incompleteness theorem requires the following crucial lemma:

$$\text{If } \alpha \text{ is a } \Sigma \text{ sentence, then } \vdash \alpha \to \mathrm{Pf}\,\ulcorner\alpha\urcorner.$$

This is an internalisation of theorem (1) above. It is proved by induction over the construction of α as a Σ formula. This requires generalising the statement above to allow the formula α to contain free variables. The technical details are complicated, and lengthy deductions in the calculus seem to be essential.

The proof sketched above incorporates numerous improvements over Gödel's original version. Gödel proved only the left-to-right direction of the equivalence (2) and required a stronger assumption than consistency, namely ω-consistency.

2.2 Hereditarily Finite Set Theory

Gödel first proved his incompleteness theorems in a first-order theory of Peano arithmetic [7]. O'Connor and Harrison do the same, while Shankar and I have both chosen a formalisation of the hereditarily finite (HF) sets. Although each theory can be formally interpreted in the other, meaning that they are of equivalent strength, the HF theory is more convenient, as it can express all set-theoretic constructions that do not require infinite sets. An HF set is a finite set of HF sets, and this recursive definition can be captured by the following three axioms:

$$z = 0 \leftrightarrow \forall x\,[x \notin z] \tag{HF1}$$

$$z = x \lhd y \leftrightarrow \forall u\,[u \in z \leftrightarrow u \in x \vee u = y] \tag{HF2}$$

$$\phi(0) \wedge \forall xy\,[\phi(x) \wedge \phi(y) \to \phi(x \lhd y)] \to \forall x\,[\phi(x)] \tag{HF3}$$

The first axiom states that 0 denotes the empty set. The second axiom defines the binary operation symbol $\lhd$ (pronounced "eats") to denote insertion into a set, so that $x \lhd y = x \cup \{y\}$. The third axiom is an induction scheme, and states that every set is created by a finite number of applications of 0 and $\lhd$.

The machine proofs of the incompleteness theorems rest on an Isabelle theory of the hereditarily finite sets. To illustrate the syntax, here are the three basic axioms as formalised in Isabelle. The type of such sets is called `hf`, and is constructed such that the axioms above can be proved.

```
lemma hempty_iff: "z=0 ⟷ (∀x. ¬ x ∈ z)"
lemma hinsert_iff: "z = x ◁ y ⟷ (∀u. u ∈ z ⟷ u ∈ x | u = y)"
lemma hf_induct_ax: "⟦P 0; ∀x. P x ⟶ (∀y. P y ⟶ P (x ◁ y))⟧ ⟹ P x"
```

The same three axioms, formalised within Isabelle as a deep embedding, form the basis for the incompleteness proofs. Type `hf` and its associated operators serve as the standard model for the embedded HF set theory.

HF set theory is equivalent to Zermelo-Frankel (ZF) set theory with the axiom of infinity negated. Many of the Isabelle definitions and theorems were taken, with minor

modifications, from Isabelle/ZF [24]. Familiar concepts such as union, intersection, set difference, power set, replacement, ordered pairing and foundation can be derived in terms of the axioms shown above [32]. A few of these derivations need to be repeated—with infinitely greater effort—in the internal calculus.

Ordinals in HF are simply natural numbers, where $n = \{0, 1, \ldots, n-1\}$. Their typical properties (for example, that they are linearly ordered) have the same proofs as in ZF set theory. Świerczkowski's proofs [32] are sometimes more elegant, and addition on ordinals is obtained through a general notion of addition of sets [15]. Finally, there are about 400 lines of material concerned with relations, functions and finite sequences. This is needed to reason about the coding of syntax for the incompleteness theorem.

2.3 Isabelle's Nominal Package

For the incompleteness theorems, we are concerned with formalising the syntax of first-order logic. Variable binding is a particular issue: it is well known that technicalities relating to bound variables and substitution have caused errors in published proofs and complicate formal treatments. O'Connor [23] reports severe difficulties in his proofs.

The *nominal* approach [6, 28] to formalising variable binding (and other sophisticated uses of variable names) is based on a theory of permutations over names. A primitive concept is *support*: $\mathrm{supp}(x)$ has a rather technical definition involving permutations, but is equivalent in typical situations to the set of free names in x. We also write $a \sharp x$ for $a \notin \mathrm{supp}(x)$, saying "$a$ is fresh for x". Isabelle's nominal package [33, 34] supports these concepts through commands such as **nominal_datatype** to introduce types, **nominal_primrec** to declare primitive recursive functions and **nominal_induct** to perform structural induction. Syntactic constructions involving variable binding are identified up to α-conversion, using a quotient construction. These mechanisms generally succeed at emulating informal standard conventions for variable names. In particular, we can usually assume that the bound variables we encounter never clash with other variables.

The best way to illustrate these ideas is by examples. The following datatype defines the syntax of terms in the HF theory:

```
nominal_datatype tm = Zero | Var name | Eats tm tm
```

The type `name` (of variable names) has been created using the nominal framework. The members of this type constitute a countable set of uninterpreted atoms. The function `nat_of_name` is a bijection between this type and the type of natural numbers.

Here is the syntax of HF formulas, which are $t \in u$, $t = u$, $\phi \vee \psi$, $\neg\phi$ or $\exists x\,[\phi]$:

```
nominal_datatype fm =
    Mem tm tm     (infixr "IN" 150)
  | Eq tm tm      (infixr "EQ" 150)
  | Disj fm fm    (infixr "OR" 130)
  | Neg fm
  | Ex x::name f::fm binds x in f
```

In most respects, this *nominal* datatype behaves exactly like a standard algebraic datatype. However, the bound variable name designated by `x` above is not significant: no function can be defined to return the name of a variable bound using `Ex`.

Substitution of a term `x` for a variable `i` is defined as follows:

```
nominal_primrec subst :: "name ⇒ tm ⇒ tm ⇒ tm"
  where
   "subst i x Zero       = Zero"
 | "subst i x (Var k)    = (if i=k then x else Var k)"
 | "subst i x (Eats t u) = Eats (subst i x t) (subst i x u)"
```

Unfortunately, most recursive definitions involving **nominal_primrec** must be followed by a series of proof steps, verifying that the function uses names legitimately. Occasionally, these proofs (omitted here) require subtle reasoning involving nominal primitives.

Substituting the term `x` for the variable `i` in the formula `A` is written `A(i::=x)`.

```
nominal_primrec  subst_fm :: "fm ⇒ name ⇒ tm ⇒ fm"
  where
    Mem:  "(Mem t u)(i::=x)  = Mem (subst i x t) (subst i x u)"
  | Eq:   "(Eq t u)(i::=x)   = Eq  (subst i x t) (subst i x u)"
  | Disj: "(Disj A B)(i::=x) = Disj (A(i::=x)) (B(i::=x))"
  | Neg:  "(Neg A)(i::=x)    = Neg (A(i::=x))"
  | Ex:   "atom j ♯ (i, x) ⟹ (Ex j A)(i::=x) = Ex j (A(i::=x))"
```

Note that the first seven cases (considering the two substitution functions collectively) are straightforward structural recursion. In the final case, we see a precondition, `atom j ♯ (i, x)`, to ensure that the substitution is legitimate within the formula `Ex j A`. There is no way to define the contrary case, where the bound variable clashes. We would have to eliminate any such clash, renaming the bound variable by applying an appropriate permutation to the formula. Thanks to the nominal framework, such explicit renaming steps are rare.

This style of formalisation is more elegant than traditional textbook definitions that do include the variable-clashing case. It is much more elegant than including renaming of the bound variable as part of the definition itself. Such "definitions" are really implementations, and greatly complicate proofs.

The commutativity of substitution (two distinct variables, each fresh for the opposite term) is easily proved.

```
lemma subst_fm_commute2 [simp]:
  "⟦atom j ♯ t; atom i ♯ u; i ≠ j⟧ ⟹ (A(i::=t))(j::=u) = (A(j::=u))(i::=t)"
  by (nominal_induct A avoiding: i j t u rule: fm.strong_induct) auto
```

The proof is by *nominal induction* on the formula `A`: the proof method for structural induction over a nominal datatype. Compared with ordinary induction, `nominal_induct` takes account of the freshness of bound variable names. The phrase `avoiding: i j t u` is the formal equivalent of the convention that when we consider the case of the existential formula `Ex k A`, the bound variable `k` can be assumed to be fresh with respect to the terms mentioned. This convention is formalised by four additional assumptions `atom k ♯ i`, `atom k ♯ j`, etc.; they ensure that substitution will be well-defined over this existential quantifier, making the proof easy.

This and many similar facts have two-step proofs, `nominal_induct` followed by `auto`. In contrast, O'Connor needed to combine three substitution lemmas (including the one above) in a giant mutual induction involving 1,900 lines of Coq. He blames the renaming step in substitution and suggests that a form of simultaneous substitution might have avoided these difficulties [23, §4.3]. An alternative, using traditional bound variable names, is to treat substitution not as a function but as a relation that holds only when no renaming is necessary. Bound variable renaming is then an independent operation. I briefly tried this idea, which allowed reasonably straightforward proofs of substitution properties, but ultimately nominal looked like a better option.

3 Theorems, Σ Formulas, Provability

The first milestone in the proof of the incompleteness theorems is the development of a first-order logical calculus equipped with enough meta-theory to guarantee that some true formulas are theorems. The previous section has already presented the definitions of the terms and formulas of this calculus. The terms are for HF set theory, and the formulas are defined via a minimal set of connectives from which others can be defined.

3.1 A Sequent Calculus for HF Set Theory

Compared with a textbook presentation, a machine development must include an effective proof system, one that can actually be used to prove non-trivial theorems.

3.1.1 Semantics

The semantics of terms and formulas are given by the obvious recursive function definitions, which yield sets and Booleans, respectively. These functions accept an environment mapping variables to values. The null environment maps all variables to 0, and is written `e0`. It involves the types `finfun` (for finite functions) [17] and `hf` (for HF sets).

```
definition e0 :: "(name, hf) finfun"     — the null environment
  where "e0 ≡ finfun_const 0"

nominal_primrec eval_tm :: "(name, hf) finfun ⇒ tm ⇒ hf"
  where
   "eval_tm e Zero = 0"
 | "eval_tm e (Var k) = finfun_apply e k"
 | "eval_tm e (Eats t u) = eval_tm e t ◁ eval_tm e u"
```

There are two things to note in the semantics of formulas. First, the special syntax ⟦`t`⟧`e` abbreviates `eval_tm e t`. Second, in the semantics of the quantifier `Ex`, note how the formula `atom k ♯ e` asserts that the bound variable, `k`, is not currently given a value by the environment, `e`.

```
nominal_primrec eval_fm :: "(name, hf) finfun ⇒ fm ⇒ bool"
  where
   "eval_fm e (t IN u) ⟷ ⟦t⟧e ∈ ⟦u⟧e"
 | "eval_fm e (t EQ u) ⟷ ⟦t⟧e = ⟦u⟧e"
 | "eval_fm e (A OR B) ⟷ eval_fm e A ∨ eval_fm e B"
 | "eval_fm e (Neg A) ⟷ (~ eval_fm e A)"
 | "atom k ♯ e ⟹ eval_fm e (Ex k A) ⟷ (∃x. eval_fm (finfun_update e k x) A)"
```

This yields the Tarski truth definition for the standard model of HF set theory. In particular, `eval_fm e0 A` denotes the truth of the sentence `A`.

3.1.2 Axioms

Świerczkowski [32] specifies a standard Hilbert-style calculus, with two rules of inference and several axioms or axiom schemes. The latter include *sentential axioms*, defining the behaviour of disjunction and negation:

```
inductive_set boolean_axioms :: "fm set"
  where
    Ident:     "A IMP A ∈ boolean_axioms"
  | DisjI1:    "A IMP (A OR B) ∈ boolean_axioms"
  | DisjCont:  "(A OR A) IMP A ∈ boolean_axioms"
  | DisjAssoc: "(A OR (B OR C)) IMP ((A OR B) OR C) ∈ boolean_axioms"
  | DisjConj:  "(C OR A) IMP (((Neg C) OR B) IMP (A OR B)) ∈ boolean_axioms"
```

Here Świerczkowski makes a tiny error, expressing the last axiom scheme as

$$(\phi \vee \psi) \wedge (\neg\phi \vee \mu) \rightarrow \psi \vee \mu.$$

Because $\wedge$ is defined in terms of $\vee$, while this axiom helps to define $\vee$, this formulation is unlikely to work. The Isabelle version eliminates $\wedge$ in favour of nested implication.

There are four primitive *equality axioms*, shown below in mathematical notation. They express reflexivity as well as substitutivity for equality, membership and the eats operator. They are not schemes but single formulas containing specific free variables. Creating an instance of an axiom for specific terms (which might involve the same variables) requires many renaming steps to insert fresh variables, before substituting for them one term at a time.

$$\begin{aligned}
&x_1 = x_1 \\
&(x_1 = x_2) \wedge (x_3 = x_4) \rightarrow [(x_1 = x_3) \rightarrow (x_2 = x_4)] \\
&(x_1 = x_2) \wedge (x_3 = x_4) \rightarrow [(x_1 \in x_3) \rightarrow (x_2 \in x_4)] \\
&(x_1 = x_2) \wedge (x_3 = x_4) \rightarrow [x_1 \lhd x_3 = x_2 \lhd x_4]
\end{aligned}$$

There is also a *specialisation* axiom scheme, of the form $\phi(t/x) \rightarrow \exists x\, \phi$:

```
inductive_set special_axioms :: "fm set" where
  I: "A(i::=x) IMP (Ex i A) ∈ special_axioms"
```

There are the axioms HF1 and HF2 for the set theory, while HF3 (*induction*) is formalised as an axiom scheme:

```
inductive_set induction_axioms :: "fm set" where
  ind:
  "atom (j::name) ♯ (i,A)
   ⟹ A(i::=Zero)
       IMP ((All i (All j (A IMP (A(i::= Var j) IMP A(i::= Eats(Var i)(Var j))))))
      IMP (All i A))
    ∈ induction_axioms"
```

Axiom schemes are conveniently introduced using **inductive_set**, simply to express set comprehensions, even though there is no actual induction.

3.1.3 Inference System

The axiom schemes shown above, along with inference rules for modus ponens and existential instantiation,[5] are combined to form the following inductive definition of theorems:

[5] From $A \rightarrow B$ infer $\exists x A \rightarrow B$, for x not free in B.

```
inductive hfthm :: "fm set ⇒ fm ⇒ bool" (infixl "⊢" 55)
  where
    Hyp:    "A ∈ H ⟹ H ⊢ A"
  | Extra:  "H ⊢ extra_axiom"
  | Bool:   "A ∈ boolean_axioms ⟹ H ⊢ A"
  | Eq:     "A ∈ equality_axioms ⟹ H ⊢ A"
  | Spec:   "A ∈ special_axioms ⟹ H ⊢ A"
  | HF:     "A ∈ HF_axioms ⟹ H ⊢ A"
  | Ind:    "A ∈ induction_axioms ⟹ H ⊢ A"
  | MP:     "H ⊢ A IMP B ⟹ H' ⊢ A ⟹ H ∪ H' ⊢ B"
  | Exists: "H ⊢ A IMP B ⟹ atom i♯B ⟹ ∀C∈H. atom i♯C ⟹ H ⊢ (Ex i A) IMP B"
```

A minor deviation from Świerczkowski is `extra_axiom`, which is abstractly specified to be an arbitrary true formula. This means that the proofs will be conducted with respect to an arbitrary finite extension of the HF theory. The first major deviation from Świerczkowski is the introduction of rule `Hyp`, with a set of assumptions. It would be virtually impossible to prove anything in his Hilbert-style proof system, and it was clear from the outset that lengthy proofs within the calculus might be necessary. Introducing `Hyp` generalises the notion of provability, allowing the development of a sort of sequent calculus, in which long but tolerably natural proofs can be constructed.

It is worth mentioning that Świerczkowski's definitions and proofs fit together very tightly, deviations often being a cause for later regret. One example, concerning an inference rule for substitution, is mentioned at the end of Sect. 4.4. Another example is that some tricks that simplify the proof of the first incompleteness theorem turn out to complicate the proof of the second.

The soundness of the calculus above is trivial to prove by induction. The deduction theorem is also straightforward, the only non-trivial case being the one for the `Exists` inference rule. The induction formula is stated as follows:

```
lemma deduction_Diff: assumes "H ⊢ B" shows "H - {C} ⊢ C IMP B"
```

This directly yields the standard formulation of the deduction theorem:

```
theorem deduction: assumes "insert A H ⊢ B" shows "H ⊢ A IMP B"
```

And this is a sequent rule for implication.

Setting up a usable sequent calculus requires much work. The corresponding Isabelle theory file, which starts with the definitions of terms and formulas and ends with a sequent formulation of the HF induction rule, is nearly 1,600 lines long. Deriving natural sequent calculus rules from the sentential and equality axioms requires lengthy chains of steps. Even in the final derived sequent calculus, equalities can only be applied one step at a time.

For another example of difficulty, consider the following definition:

```
definition Fls  where "Fls ≡ Zero IN Zero"
```

Proving that `Fls` has the properties of falsehood is surprisingly tricky. The relevant axiom, HF1, is formulated using universal quantifiers, which are defined as negated existentials; deriving the expected properties of universal quantification seems to require something like `Fls` itself.

The derived sequent calculus has specialised rules to operate on conjunctions, disjunctions, etc., in the hypothesis part of a sequent. They are crude, but good enough. Used with Isabelle's automatic tactics, they ease somewhat the task of constructing formal HF proofs. Users can extend Isabelle with proof procedures coded in ML, and better automation for the calculus might thereby be achieved. At the time, such a side-project did not seem to be worth the effort.

3.2 A Formal Theory of Functions

Recursion is not available in HF set theory, and recursive functions must be constructed explicitly. Each recursive computation is expressed in terms of the existence of a sequence $(s_i)_{i \le k}$ such that s_i is related to s_m and s_n for m, $n < i$. Moreover, a sequence is formally a relation rather than a function. In the metalanguage, we write `app s k` for s_k, governed by the theorem

lemma *app_equality: "hfunction s ⟹ ⟨x,y⟩ ∈ s ⟹ app s x = y"*

The following two functions express the recursive definition of sequences, as needed for the Gödel development:

```
"Builds B C s l ≡ B(app s l) ∨ (∃m∈l. ∃n∈l. C(app s l) (app s m) (app s n))"
"BuildSeq B C s k y ≡ LstSeq s k y ∧ (∀l∈succ k. Builds B C s l)"
```

The statement `Builds B C s l` constrains element `l` of sequence `s`, namely `app s l`. We have either `B(app s l)`, or `C (app s l) (app s m) (app s n))` where `m∈l` and `n∈l`. For the natural numbers, set membership coincides with the less-than relation. Therefore, we are referring to a sequence s and element s_l where either the base case $B(s_l)$ holds, or else the recursive step $C(s_l, s_m, s_n)$ for m, $n < l$. The statement `BuildSeq B C s k y` states that the sequence `s` has been constructed in this way right up to the value `app s k`, or in other words, s_k, where $y = s_k$.

To formalise the basis for this approach requires a series of definitions in the HF calculus, introducing the subset relation, ordinals (which are simply natural numbers), ordered pairs, relations with a given domain, etc. Foundation (the well-foundedness of the membership relation) must also be proved, which in turn requires additional definitions. A few highlights are shown below.

The *subset relation* is defined, with infix syntax `SUBS`, with the help of `All2`, the bounded universal quantifier.

```
nominal_primrec Subset :: "tm ⇒ tm ⇒ fm"   (infixr "SUBS" 150)
  where "atom z ♯ (t, u) ⟹ t SUBS u = All2 z t ((Var z) IN u)"
```

In standard notation, this says $t \subseteq u = (\forall z \in t)[z \in u]$. The definition uses **nominal_primrec**, even though it is not recursive, because it requires `z` to be fresh with respect to the terms `t` and `u`, among other nominal-related technicalities.

Extensionality is taken as an axiom in traditional set theories, but in HF it can be proved by induction. However, many straightforward properties of the subset relation must first be derived.

lemma *Extensionality: "H ⊢ x EQ y IFF x SUBS y AND y SUBS x"*

Ordinals will be familiar to set theorists. The definition is the usual one, and shown below mainly as an example of a slightly more complicated HF formula. Two variables, `y` and `z`, must be fresh for each other and `x`.

```
nominal_primrec OrdP :: "tm ⇒ fm"
  where "⟦atom y ♯ (x, z); atom z ♯ x⟧ ⟹
    OrdP x = All2 y x ((Var y) SUBS x  AND
             All2 z (Var y) ((Var z) SUBS (Var y)))"
```

The formal definition of a *function* (as a single-valued set of pairs) is subject to several complications. As we shall see in Sect. 3.3 below, all definitions must use Σ formulas, which requires certain non-standard formulations. In particular, $x \neq y$ is not a Σ formula in general, but it can be expressed as $x < y \vee y < x$ if x and y are ordinals. The following primitive is used extensively when coding the syntax of HF within itself.

```
nominal_primrec LstSeqP :: "tm ⇒ tm ⇒ tm ⇒ fm"
  where
   "LstSeqP s k y = OrdP k AND HDomain_Incl s (SUCC k) AND
                    HFun_Sigma s AND HPair k y IN s"
```

Informally, `LstSeqP s k y` means that s is a non-empty sequence whose domain includes the set $\{0, \ldots, k\}$ (which is the ordinal $k + 1$: the sequence is at least that long). Moreover, $y = s_k$; that would be written $\langle k,y\rangle \in s$ in the metalanguage, but becomes `HPair k y IN s` in the HF calculus, as seen above.

Świerczkowski [32] prefers slightly different definitions, specifying the domain to be exactly k, where $k > 0$ and $y = s_{k-1}$. The definition shown above simplifies the proof of the first incompleteness theorem, but complicates the proof of the second, in particular because they allow a sequence to be longer than necessary.

This part of the development consists mainly of proofs in the HF calculus, and is nearly 1,300 lines long.

3.3 Σ Formulas and Provability

Gödel had the foresight to recognise the value of minimising the need to write explicit formal proofs, without relying on the assumption that certain proofs could "obviously" be formalised. Instead, he developed enough meta-theory to prove that these proofs existed. One approach for this [2, 32] relies on the concept of Σ formulas. These are inductively defined to include all formulas of the form $t \in u$, $t = u$, $\alpha \vee \beta$, $\alpha \wedge \beta$, $\exists x\, \alpha$ and $(\forall x \in t)\, \alpha$. (These are closely related to the Σ_1 formulas of the arithmetical hierarchy.) It follows by induction on this construction that every true Σ sentence has a formal proof. Intuitively, the reasoning is that the atomic cases can be calculated, the Boolean cases can be done recursively, and the bounded universal quantifier can be replaced by a finite conjunction. The existential case holds because the semantics of $\exists x\, \alpha$ yields a specific witnessing value, again allowing an appeal to the induction hypothesis.

The Σ formula approach is a good fit to the sort of formulas used in the coding of syntax. In these formulas, universal quantifiers have simple upper bounds, typically a variable giving the length of a sequence, while existential variables are unbounded. Gödel's original proofs required all quantifiers to be bounded. Existential quantifiers were bounded by complicated expressions requiring deep and difficult arithmetic justifications. Boolos presents similar material in a more modern form [2, p. 41]. Relying exclusively on Σ formulas avoids these complications, but instead some straightforward properties have to be proven formally in the HF calculus.

A complication is that proving the second incompleteness theorem requires another induction over Σ formulas. To minimise that proof effort, it helps to use as restrictive a definition as possible. The *strict* Σ formulas consist of all formulas of the form $x \in y$, $\alpha \vee \beta$, $\alpha \wedge \beta$, $\exists x\, \alpha$ and $(\forall x \in y)\, \alpha$. Here, x and y are not general terms, but variables. We further stipulate y not free in α in $(\forall x \in y)\, \alpha$; then in the main induction leading to the second incompleteness theorem, Case 2 of Lemma 9.7 [32] can be omitted.

```
inductive ss_fm :: "fm ⇒ bool" where
    MemI:  "ss_fm (Var i IN Var j)"
  | DisjI: "ss_fm A ⟹ ss_fm B ⟹ ss_fm (A OR B)"
  | ConjI: "ss_fm A ⟹ ss_fm B ⟹ ss_fm (A AND B)"
  | ExI:   "ss_fm A ⟹ ss_fm (Ex i A)"
  | All2I: "ss_fm A ⟹ atom j ♯ (i,A) ⟹ ss_fm (All2 i (Var j) A)"
```

Now, a Σ formula is by definition one that is provably equivalent (in HF) to some strict Σ formula containing no additional free variables. In another minor oversight, Świerczkowski omits the free variable condition, but it is necessary.

```
definition Sigma_fm :: "fm ⇒ bool"
  where "Sigma_fm A ⟷ (∃B. ss_fm B ∧ supp B ⊆ supp A ∧ {} ⊢ A IFF B)"
```

As always, Świerczkowski's exposition is valuable, but far from complete. Showing that Σ formulas include $t \in u$, $t = u$ and $(\forall x \in t)\,\alpha$ for all terms t and u (and not only for variables) is far from obvious. These necessary facts are not even stated clearly. A crucial insight is to focus on proving that $t \in u$ and $t \subseteq u$ are Σ formulas. Consideration of the cases $t \in 0$, $t \in u_1 \lhd u_2$, $0 \subseteq u$, $t_1 \lhd t_2 \subseteq u$ shows that each reduces to false, true or a combination of simpler uses of $\in$ or $\subseteq$. This observation suggests proving that $t \in u$ and $t \subseteq u$ are Σ formulas by mutual induction on the combined sizes of t and u.

```
lemma Subset_Mem_sf_lemma:
  "size t + size u < n ⟹ Sigma_fm (t SUBS u) ∧ Sigma_fm (t IN u)"
```

The identical argument turns out to be needed for the second incompleteness theorem itself, formalised this time within the HF calculus. This coincidence should not be that surprising, as it is known that the second theorem could in principle be shown by formalising the first theorem within its own calculus.

Now that we have taken care of $t \subseteq u$, proving that $t = u$ is a Σ formula is trivial by extensionality, and the one remaining objective is $(\forall x \in t)\,\alpha$. But with equality available, we can reduce this case to the strict Σ formula $(\forall x \in y)\,\alpha$ with the help of a lemma:

```
lemma All2_term_Iff: "atom i ♯ t ⟹ atom j ♯ (i,t,A) ⟹
                      {} ⊢ (All2 i t A) IFF Ex j (Var j EQ t AND All2 i (Var j) A)"
```

This is simply $(\forall x \in t)\,A \leftrightarrow \exists y\,[y = t \wedge (\forall x \in y)\,A]$ expressed in the HF calculus, where it is easily proved. We could prove that $(\forall x \in t)\,\alpha$ is a Σ formula by induction on t, but this approach leads to complications.

Virtually all predicates defined for the Gödel development are carefully designed to take the form of Σ formulas. Here are two examples; most such lemmas hold immediately by the construction of the given formula.

```
lemma Subset_sf: "Sigma_fm (t SUBS u)"
lemma LstSeqP_sf: "Sigma_fm (LstSeqP t u v)"
```

The next milestone asserts that if α is a ground Σ formula (and therefore a sentence) and α evaluates to true, then α is a theorem. The proof is by induction on the size of the formula, and then by case analysis on its outer form. The case $t \in u$ falls to a mutual induction with $t \subseteq u$ resembling the one shown above. The case $(\forall x \in t)\,\alpha$ is effectively expanded to a conjunction.

```
theorem Sigma_fm_imp_thm: "⟦Sigma_fm α; ground_fm α; eval_fm e0 α⟧ ⟹ {} ⊢ α"
```

Every true Σ sentence is a theorem. This crucial meta-theoretic result is used eight times in the development. Without it, gigantic explicit HF proofs would be necessary.

4 Coding Provability in HF Within Itself

The key insight leading to the proof of Gödel's theorem is that a sufficiently strong logical calculus can represent its syntax within itself, and in particular, the property of a given formula being provable. This task divides into three parts: coding the syntax, defining predicates to describe the coding and finally, defining predicates to describe the inference system.

4.1 Coding Terms, Formulas, Abstraction and Substitution

In advocating the use of HF over PA, Świerczkowski begins by emphasising the ease of coding syntax:

> It is at hand to code the variables x_1, x_2, ... simply by the ordinals 1, 2, ... The constant 0 can be coded as 0, and the remaining 6 symbols as n-tuples of 0s, say $\in$ as $\langle 0, 0\rangle$, etc. And here ends the arbitrariness of coding, which is so unpleasant when languages are arithmetized. [32, p. 5]

The adequacy of these definitions is easy to prove in HF itself. The full list is as follows: $\ulcorner 0 \urcorner = 0$, $\ulcorner x_i \urcorner = i + 1$, $\ulcorner \in \urcorner = \langle 0, 0\rangle$, $\ulcorner \lhd \urcorner = \langle 0, 0, 0\rangle$, $\ulcorner = \urcorner = \langle 0, 0, 0, 0\rangle$, $\ulcorner \vee \urcorner = \langle 0, 0, 0, 0, 0\rangle$, $\ulcorner \neg \urcorner = \langle 0, 0, 0, 0, 0, 0\rangle$, $\ulcorner \exists \urcorner = \langle 0, 0, 0, 0, 0, 0, 0\rangle$. We have a few differences from Świerczkowski: $\ulcorner x_i \urcorner = i + 1$ because our variables start at zero, and for the kth de Bruijn index we use $\langle\langle 0, 0, 0, 0, 0, 0, 0, 0\rangle, k\rangle$. Obviously $\in$ means nothing by itself, so $\ulcorner \in \urcorner = \langle 0, 0\rangle$ really means $\ulcorner t \in u \urcorner = \langle\langle 0, 0\rangle, \ulcorner t \urcorner, \ulcorner u \urcorner\rangle$, etc. Note that nests of n-tuples terminated by ordinals can be decomposed uniquely.

De Bruijn equivalents of terms and formulas are then declared. To repeat: de Bruijn syntax is used for coding, for which it is ideal, allowing the simplest possible definitions of abstraction and substitution. Although it destroys readability, encodings are never readable anyway. Using nominal here is out of the question. The entire theory of nominal Isabelle would need to be formalised within the embedded calculus. Quite apart from the work involved, the necessary equivalence classes would be infinite sets, which are not available in HF.

The strongest argument for HF is that the mathematical basis of its coding scheme is simply ordered pairs defined in the standard set-theoretic way. An elementary formal argument justifies this. In contrast, the usual arithmetic encoding relies on either the Chinese remainder theorem or unique prime factorisation. This fragment of number theory would have to be formalised within the embedded calculus in order to reason about encoded formulas, which is necessary to prove the second incompleteness theorem. It must be emphasised that proving anything in the calculus (where such luxuries as a simplifier, recursion and even function symbols are not available) is much more difficult than proving the same result in a proof assistant.

4.1.1 Introducing de Bruijn Terms and Formulas

De Bruijn terms resemble the type `tm` declared in Sect. 2.3, but include the `DBInd` constructor for bound variable indices as well as the `DBVar` constructor for free variables.

```
nominal_datatype dbtm = DBZero | DBVar name | DBInd nat | DBEats dbtm dbtm
```

De Bruijn formula contructors involve no explicit variable binding, creating an apparent similarity between `DBNeg` and `DBEx`, although the latter creates an implicit variable binding scope.

```
nominal_datatype dbfm =
    DBMem dbtm dbtm
  | DBEq dbtm dbtm
  | DBDisj dbfm dbfm
  | DBNeg dbfm
  | DBEx dbfm
```

How this works should become clear as we consider how terms and formulas are translated into their de Bruijn equivalents. To begin with, we need a lookup function taking a list of names (representing variables bound in the current context, innermost first) and a name to be looked up. The integer `n`, initially 0, is the index to substitute if the name is next in the list.

```
fun lookup :: "name list ⇒ nat ⇒ name ⇒ dbtm"
  where
    "lookup [] n x = DBVar x"
  | "lookup (y # ys) n x = (if x = y then DBInd n else (lookup ys (Suc n) x))"
```

To translate a term into de Bruijn format, the key step is to resolve name references using `lookup`. Names bound in the local environment are replaced by the corresponding indices, while other names are left as they were.

```
nominal_primrec trans_tm :: "name list ⇒ tm ⇒ dbtm"
  where
   "trans_tm e Zero = DBZero"
 | "trans_tm e (Var k) = lookup e 0 k"
 | "trans_tm e (Eats t u) = DBEats (trans_tm e t) (trans_tm e u)"
```

Noteworthy is the final case of `trans_fm`, which requires the bound variable `k` in the quantified formula `Ex k A` to be fresh with respect to `e`, our list of previously-encountered bound variables. In the recursive call, `k` is added to the list, which therefore consists of distinct names.

```
nominal_primrec trans_fm :: "name list ⇒ fm ⇒ dbfm"
  where
   "trans_fm e (Mem t u) = DBMem (trans_tm e t) (trans_tm e u)"
 | "trans_fm e (Eq t u)  = DBEq (trans_tm e t) (trans_tm e u)"
 | "trans_fm e (Disj A B) = DBDisj (trans_fm e A) (trans_fm e B)"
 | "trans_fm e (Neg A)   = DBNeg (trans_fm e A)"
 | "atom k ♯ e ⟹ trans_fm e (Ex k A) = DBEx (trans_fm (k#e) A)"
```

Syntactic operations for de Bruijn notation tend to be straightforward, as there are no bound variable names that might clash. Comparisons with previous formalisations of the λ-calculus may be illuminating, but the usual lifting operation [18, 21] is unnecessary. That is because the HF calculus does not allow reductions anywhere, as in the λ-calculus. Substitutions only happen at the top level and never within deeper bound variable contexts. For us, substitution is the usual operation of replacing a variable by a term, which contains no bound variables. (Substitution can alternatively be defined to replace a de Bruijn index by a term.)

The special de Bruijn operation is *abstraction*. This replaces every occurrence of a given free variable in a term or formula by a de Bruijn index, in preparation for binding. For example, abstracting the formula `DBMem (DBVar x) (DBVar y)` over the variable `y` yields `DBMem (DBVar x) (DBInd 0)`. This is actually ill-formed, but attaching a quantifier yields the de Bruijn formula

```
DBEx (DBMem (DBVar x) (DBInd 0)),
```

representing $\exists y\,[x \in y]$. Abstracting this over the free variable `x` and attaching another quantifier yields

```
DBEx (DBEx (DBMem (DBInd 1) (DBInd 0))),
```

which is the formula $\exists xy\,[x \in y]$. An index of 1 has been substituted in order to skip over the inner binder.

4.1.2 Well-Formed de Bruijn Terms and Formulas

With the de Bruijn approach, an index of 0 designates the innermost enclosing binder, while an index of 1 designates the next-innermost binder, etc. (Here, the only binder is `DBEx`.) If every index has a matching binder (the index i must be nested within at least $i + 1$ binders), then the term or formula is *well-formed*. Recall the examples of abstraction above, where a binder must be attached afterwards.

In particular, as our terms do not contain any binding constructs, a *well-formed term* may contain no de Bruijn indices. In contrast to more traditional notions of logical syntax, if you take a well-formed formula and view one of its subformulas or subterms in isolation, it will not necessarily be well-formed. The situation is analogous to extracting a fragment of a program, removing it from necessary enclosing declarations.

The property of being a well-formed de Bruijn term or formula is defined inductively. The syntactic predicates defined below recognise such well-formed formulas. A well-formed de Bruijn term has no indices (`DBInd`) at all:

```
inductive wf_dbtm :: "dbtm ⇒ bool"
  where
    Zero:  "wf_dbtm DBZero"
  | Var:   "wf_dbtm (DBVar name)"
  | Eats:  "wf_dbtm t1 ⟹ wf_dbtm t2 ⟹ wf_dbtm (DBEats t1 t2)"
```

A trivial induction shows that for every well-founded de Bruijn term there is an equivalent standard term. The only cases to be considered (as per the definition above) are `Zero`, `Var` and `Eats`.

```
lemma wf_dbtm_imp_is_tm:
  assumes "wf_dbtm x"
  shows "∃t::tm. x = trans_tm [] t"
```

A well-formed de Bruijn formula is constructed from other well-formed terms and formulas, and indices can only be introduced by applying abstraction (`abst_dbfm`) over a given `name` to another well-formed formula, in the existential case. Specifically, the `Ex` clause below states that, starting with a well-formed formula `A`, abstracting over some `name` and applying `DBEx` to the result yields another well-formed formula.

```
inductive wf_dbfm :: "dbfm ⇒ bool"
  where
    Mem:   "wf_dbtm t1 ⟹ wf_dbtm t2 ⟹ wf_dbfm (DBMem t1 t2)"
  | Eq:    "wf_dbtm t1 ⟹ wf_dbtm t2 ⟹ wf_dbfm (DBEq t1 t2)"
  | Disj:  "wf_dbfm A1 ⟹ wf_dbfm A2 ⟹ wf_dbfm (DBDisj A1 A2)"
  | Neg:   "wf_dbfm A ⟹ wf_dbfm (DBNeg A)"
  | Ex:    "wf_dbfm A ⟹ wf_dbfm (DBEx (abst_dbfm name 0 A))"
```

This definition formalises the allowed forms of construction, rather than stating explicitly that every index must have a matching binder.

A refinement must be mentioned. *Strong* nominal induction (already seen above, Sect. 2.3) formalises the assumption that bound variables revealed by induction can be assumed not to clash with other variables. This is set up automatically for nominal datatypes, but here requires a manual step. The command **nominal_inductive** sets up strong induction for `name` in the `Ex` case of the inductive definition above; we must prove that `name` is not significant according to the nominal theory, and then get to assume that `name` will not clash. This step (details omitted) seems to be necessary in order to complete some inductive proofs about `wf_dbfm`.

4.1.3 Quoting Terms and Formulas

It is essential to remember that Gödel encodings are terms (having type `tm`), not sets or numbers. Textbook presentations identify terms with their denotations for the sake of clarity, but this can be confusing. The undecidable formula contains an encoding of itself in the form of a term. First, we must define codes for de Bruijn terms and formulas.

```
function quot_dbtm :: "dbtm ⇒ tm"
  where
   "quot_dbtm DBZero = Zero"
 | "quot_dbtm (DBVar name) = ORD_OF (Suc (nat_of_name name))"
 | "quot_dbtm (DBInd k)    = HPair (HTuple 6) (ORD_OF k)"
 | "quot_dbtm (DBEats t u) = HPair (HTuple 1) (HPair (quot_dbtm t) (quot_dbtm u))"
```

The codes of real terms and formulas (for which we set up the overloaded $\lceil\cdots\rceil$ syntax) are obtained by first translating them to their de Bruijn equivalents and then encoding. We finally obtain facts such as the following:

```
lemma quot_Zero: "⌈Zero⌉ = Zero"
lemma quot_Var:  "⌈Var x⌉ = SUCC (ORD_OF (nat_of_name x))"
lemma quot_Eats: "⌈Eats x y⌉ = HPair (HTuple 1) (HPair ⌈x⌉ ⌈y⌉)"
lemma quot_Eq:   "⌈x EQ y⌉ = HPair (HTuple 2) (HPair (⌈x⌉) (⌈y⌉))"
lemma quot_Disj: "⌈A OR B⌉ = HPair (HTuple 3) (HPair (⌈A⌉) (⌈B⌉))"
lemma quot_Ex:   "⌈Ex i A⌉ = HPair (HTuple 5) (quot_dbfm (trans_fm [i] A))"
```

Note that `HPair` constructs an HF term denoting a pair, while `HTuple` n constructs an $(n+2)$-tuple of zeros. Proofs often refer to the denotations of terms rather than to the terms themselves, so the functions `q_Eats`, `q_Mem`, `q_Eq`, `q_Neg`, `q_Disj`, `q_Ex` are defined to express these codes. Here are some examples:

```
"q_Var i ≡ succ (ord_of (nat_of_name i))"
"q_Eats x y ≡ ⟨htuple 1, x, y⟩"
"q_Disj x y ≡ ⟨htuple 3, x, y⟩"
"q_Ex x ≡ ⟨htuple 5, x⟩"
```

Note that ⟨`x,y`⟩ denotes the pair of `x` and `y` as sets, in other words, of type `hf`.

4.2 Predicates for the Coding of Syntax

The next and most arduous step is to define logical predicates corresponding to each of the syntactic concepts (terms, formulas, abstraction, substitution) mentioned above. Textbooks and articles describe each predicate at varying levels of detail. Gödel [8] gives full definitions, as does Świerczkowski. Boolos [2] gives many details of how coding is set up, and gives the predicates for terms and formulas, but not for any operations upon them. Hodel [13], like many textbook authors, relies heavily on "algorithms" written in English. The definitions indeed amount to pages of computer code. Authors typically conclude with a "theorem" asserting the correctness of this code. For example, Świerczkowski [32, Sect. 3–4] presents 34 highly technical definitions, justified by seven lines of proof.

Proving the correctness of this lengthy series of definitions requires a substantial effort, and the proofs (being syntactically oriented) are tiresome. It is helpful to introduce shadow versions of all predicates in Isabelle/HOL's native logic, as well as in HF. Having two versions of each predicate simplifies the task of relating the HF version of

the predicate to the syntactic concept that it is intended to represent; the first step is to prove that the HF formula is equivalent to the syntactically similar definition written in Isabelle's higher-order logic, which then is proved to satisfy deeper properties. The shadow predicates also give an easy way to refer to the truth of the corresponding HF predicate; because each is defined to be a Σ formula, that gives a quick way (using theorem *Sigma_fm_imp_thm* above) to show that some ground instance of the predicate can be proved formally in HF. Also, one way to arrive at the correct definition of an HF predicate is to define its shadow equivalent first, since proving that it implies the required properties is much easier in Isabelle/HOL's native logic than in HF.

Świerczkowski [32] defines a full set of syntactic predicates, leaving nothing as an exercise. Unfortunately, the introduction of de Bruijn syntax necessitates rewriting many of these definitions. Some predicates (such as the variable occurrence test) are replaced by others (abstraction over a variable occurrence). The final list includes predicates to recognise the following items:

- the codes of well-formed terms (and constant terms, without variables)
- correct instances of abstraction (of a term or formula) over a variable
- correct instances of substitution (in a term or formula) for a variable
- the codes of well-formed formulas

As explained below, abstraction over a formula needs to be defined before the notion of a formula itself. We also need the property of variable non-occurrence, "x does not occur in A", which can be expressed directly as "substituting 0 for x in A yields A". This little trick eliminates the need for a full definition.

Each operation is first defined in its sequence form (expressing that sequence s is built up in an appropriate way and that s_k is a specific value); existential quantification over s and k then yields the final predicate. Formalising the sequence of steps is a primitive way to express recursion. Moreover, it tends to yield Σ formulas.

The simplest example is the predicate for constants. The shadow predicate can be defined with the help of *BuildSeq*, mentioned in Sect. 3.2 above. Note that shadow predicates are written in ordinary higher-order logic, and refer to syntactic codes using set values. We see below that in the sequence buildup, each element is either 0 (which is the code of the constant zero) or else has the form *q_Eats v w*, which is the code for $v \lhd w$.

```
definition SeqConst :: "hf ⇒ hf ⇒ hf ⇒ bool"
  where "SeqConst s k t ≡ BuildSeq (λu. u=0) (λu v w. u = q_Eats v w) s k t"
```

Thus a constant expression is built up from 0 using the $\lhd$ operator. The idea is that every element of the sequence is either 0 or has the form $\ulcorner x \lhd y \urcorner$, where x and y occur earlier in the sequence. Most of the other syntactic predicates fit exactly the same pattern, but with different base cases and constructors. A function must be coded as a relation, and a typical base case might be $\langle 0, 0\rangle$, other sequence elements having the form $\langle \ulcorner x \lhd y \urcorner, \ulcorner x' \lhd y' \urcorner\rangle$, where $\langle x, x'\rangle$ and $\langle y, y'\rangle$ occur earlier in the sequence. Substitution is codified in this manner. A function taking two arguments is coded as a sequence of triples, etc.

The discussion above relates to shadow predicates, which define formulas of Isabelle/HOL relating HF sets. The real predicates, which denote formulas of the HF calculus, are based on exactly the same ideas except that the various set constructions must be expressed using the HF term language. Note that the real predicates typically have names ending with P.

The following formula again specifies the notion of a constant term. It is simply the result of expressing the definition of *SeqConst* using HF syntax, expanding the definition of *BuildSeq*. The syntactic sugar for a reference to a sequence element s_m within some formula ϕ must now be expanded to its true form: $\phi(s_m)$ becomes $\exists y\,[\langle m, y\rangle \in s \wedge \phi(y)]$.

```
nominal_primrec SeqConstP :: "tm ⇒ tm ⇒ tm ⇒ fm"
  where "⟦atom l ♯ (s,k,sl,m,n,sm,sn);  atom sl ♯ (s,m,n,sm,sn);
          atom m ♯ (s,n,sm,sn);  atom n ♯ (s,sm,sn);
          atom sm ♯ (s,sn);  atom sn ♯ (s)⟧ ⟹
    SeqConstP s k t =
      LstSeqP s k t AND
      All2 l (SUCC k) (Ex sl (HPair (Var l) (Var sl) IN s AND (Var sl EQ Zero OR
             Ex m (Ex n (Ex sm (Ex sn (Var m IN Var l AND Var n IN Var l AND
                 HPair (Var m) (Var sm) IN s AND HPair (Var n) (Var sn) IN s AND
                 Var sl EQ Q_Eats (Var sm) (Var sn))))))))"
```

We have five bound variables, namely *l*, *sl*, *m*, *sm*, *n*, *sn*, which must be constrained to be distinct from one another using the freshness conditions shown. This nominal boilerplate may seem excessive. However, to define this predicate without nominal syntax, bound variable names might have to be calculated, perhaps by taking the maximum of the bound variables in *s*, *k* and *t* and continuing from there. Nominal constrains the variables more abstractly and flexibly.

As mentioned above, sometimes we deal with sequences of pairs or triples, with correspondingly more complicated formulas. Where a predicate describes a function such as substitution, the sequence being built up consists of ordered pairs of arguments and results, and there are typically nine bound variables. To perform abstraction over a formula requires keeping track of the depth of quantifier nesting during recursion. This is a two-argument function, so the sequence being built up consists of ordered triples and there are 12 bound variables. Although the nominal system copes with these complicated expressions, the processing time can be measured in tens of seconds.

Now that we have defined the buildup of a sequence of constants, a constant itself is trivial. The existence of any properly formed sequence *s* of length *k* culminating with some term *t* guarantees that *t* is a constant term. Here are both the shadow and HF calculus versions of the predicate.

```
definition Const :: "hf ⇒ bool"
  where "Const t ≡ (∃ s k. SeqConst s k t)"
nominal_primrec ConstP :: "tm ⇒ fm"
  where "⟦atom k ♯ (s,t); atom s ♯ t⟧ ⟹
    ConstP t = Ex s (Ex k (SeqConstP (Var s) (Var k) t))"
```

Why don't we define the HF predicate *BuildSeqP* analogously to *BuildSeq*, which expresses the definition of *SeqConst* so succinctly? Then we might expect to avoid the mess above, defining a predicate such as *SeqConstP* in a single line. This was actually attempted, but the nominal system is not really suitable for formalising higher-order definitions. Complicated auxiliary definitions and proofs are required. It is easier simply to write out the definitions, especially as their very repetitiveness allows proof development by cut and paste.

One tiny consolidation has been done. We need to define the predicates *Term* and *TermP* analogously to *Const* and *ConstP* above but allowing variables. The question of whether variables are allowed in a term or not can be governed by a Boolean. The proof development therefore introduces the predicate *SeqCTermP*, taking a Boolean argument, from which *SeqTermP* and *SeqConstP* are trivially obtained.

```
abbreviation SeqTermP :: "tm ⇒ tm ⇒ tm ⇒ fm"
  where "SeqTermP ≡ SeqCTermP True"

abbreviation SeqConstP :: "tm ⇒ tm ⇒ tm ⇒ fm"
  where "SeqConstP ≡ SeqCTermP False"
```

In this way, we can define the very similar predicates *TermP* and *ConstP* with a minimum of repeated material.

Many other predicates must be defined. Abstraction and substitution must be defined separately for terms, atomic formulas and formulas. Here are the shadow definitions of abstraction and substitution for terms. They are similar enough (both involve replacing a variable) that a single function, *SeqStTerm*, can express both. *BuildSeq2* is similar to *BuildSeq* above, but constructs a sequence of pairs.

```
definition SeqStTerm :: "hf ⇒ hf ⇒ hf ⇒ hf ⇒ hf ⇒ hf ⇒ bool"
where "SeqStTerm v u x x' s k ≡
   is_Var v ∧ BuildSeq2 (λy y'. (is_Ind y ∨ Ord y) ∧ y' = (if y=v then u else y))
                (λu u' v v' w w'. u = q_Eats v w ∧ u' = q_Eats v' w') s k x x'"

definition AbstTerm :: "hf ⇒ hf ⇒ hf ⇒ hf ⇒ bool"
  where "AbstTerm v i x x' ≡ Ord i ∧ (∃s k. SeqStTerm v (q_Ind i) x x' s k)"

definition SubstTerm :: "hf ⇒ hf ⇒ hf ⇒ hf ⇒ bool"
  where "SubstTerm v u x x' ≡ Term u ∧ (∃s k. SeqStTerm v u x x' s k)"
```

Abstraction over formulas (*AbstForm*/*AbstFormP*) must be defined before formulas themselves, in order to formalise existential quantification. There is no circularity here: the abstraction operation can be defined independently of the notion of a well-formed formula, and is not restricted to them. The definition involves sequences of triples and is too complicated to present here.

With abstraction over formulas defined, we can finally define the concept of a formula itself. An *Atomic* formula involves two terms, combined using the relations *EQ* or *IN*. Then *MakeForm* combines one or two existing formulas to build more complex ones. It constrains *y* to be the code of a formula constructed from existing formulas *u* and *v* by the disjunction $u \vee v$, the negation $\neg u$ or the existential formula $\exists(u')$, where *u'* has been obtained by abstracting *u* over some variable, *v* via the predicate *AbstForm*.

```
definition MakeForm :: "hf ⇒ hf ⇒ hf ⇒ bool"
  where "MakeForm y u w ≡
           y = q_Disj u w ∨ y = q_Neg u ∨
           (∃v u'. AbstForm v 0 u u' ∧ y = q_Ex u')"

nominal_primrec MakeFormP :: "tm ⇒ tm ⇒ tm ⇒ fm"
  where "⟦atom v ♯ (y,u,w,au); atom au ♯ (y,u,w)⟧ ⟹
    MakeFormP y u w =
      y EQ Q_Disj u w OR y EQ Q_Neg u OR
      Ex v (Ex au (AbstFormP (Var v) Zero u (Var au) AND y EQ Q_Ex (Var au)))"
```

Now, the sequence buildup of a formula can be defined with *Atomic* covering the base case and *MakeForm* expressing one level of the construction. Using similar methods to those illustrated above for constant terms, we arrive at the shadow predicate *Form* and the corresponding HF predicate *FormP*.

4.3 Verifying the Coding Predicates

Most textbook presentations take the correctness of such definitions as obvious, and indeed many properties are not difficult to prove. To show that the predicate *Term*

accepts all coded terms, a necessary lemma is to show the analogous property for well-formed de Bruijn terms:

```
lemma wf_Term_quot_dbtm:
  assumes "wf_dbtm u" shows "Term ⟦quot_dbtm u⟧e"
```

The proof is by induction on the construction of `u` (in other words, on the inductive definition of `wf_dbtm u`), and is routine by the definitions of the predicates `Term` and `SeqTerm`. This implies the desired result for terms, by the (overloaded) definition of ⌜`t`⌝.

```
corollary Term_quot_tm: fixes t::tm  shows "Term ⟦⌜t⌝⟧e"
```

Note that both results concern the shadow predicate `Term`, not the HF predicate `TermP`. The argument of `Term` is a set, denoted using the evaluation operator, ⟦...⟧e. Direct proofs about HF predicates are long and tiresome. Fortunately, many such questions can be reduced to the corresponding questions involving shadow predicates, because codes are ground terms; then the theorem `Sigma_fm_imp_thm` guarantees the existence of a proof, sparing us the need to construct one.

The converse correctness property must also be proved, namely that everything accepted by `Term` actually is the code of some term. The proof requires a lemma about the predicate `SeqTerm`. The reasoning is simply that constants and variables are well-formed, and that combining two well-formed terms preserves this property. Such proofs are streamlined through the use of `BuildSeq_induct`, a derived rule for reasoning about sequence construction by induction on the length of the sequence.

```
lemma Term_imp_wf_dbtm:
  assumes "Term x" obtains t::dbtm where "wf_dbtm t" "x = ⟦quot_dbtm t⟧e"
```

By the meaning of **obtains**, we see that if `Term x` then there exists some well-formed de Bruijn term `t` whose code evaluates to `x`. Since for every well-formed de Bruijn term there exists an equivalent standard term of type `tm`, we can conclude that `Term x` implies that `x` is the code of some term.

```
corollary Term_imp_is_tm:
  assumes "Term x" obtains t::tm where "x = ⟦⌜t⌝⟧ e"
```

Similar theorems—with similar proofs—are necessary for each of the syntactic predicates. For example, the following result expresses that `SubstForm` correctly models the result `A(i::=t)` of substituting `t` for `i` in the formula `A`.

```
lemma SubstForm_quot_unique:
  "SubstForm (q_Var i) ⟦⌜t⌝⟧e ⟦⌜A⌝⟧e x' ⟷ x' = ⟦⌜A(i::=t)⌝⟧e"
```

4.4 Predicates for the Coding of Deduction

On the whole, the formalisation of deduction is quite different from the formalisation of syntactic operations, which mostly involve simulated recursion over terms or formulas. A proof step in the HF calculus is an axiom, an axiom scheme or an inference rule. Axioms and propositional inference rules are straightforward to recognise using the existing syntactic primitives. Simply `x EQ` ⌜`refl_ax`⌝ tests whether `x` denotes the reflexivity axiom. More complicated are inference rules involving quantification, where several syntactic conditions including abstraction and substitution need to be checked in sequence. For example, consider *specialisation axioms* of the form $\phi(t/x) \to \exists x\, \phi$.

```
nominal_primrec Special_axP :: "tm ⇒ fm" where
  "⟦atom v ♯ (p,sx,y,ax,x); atom x ♯ (p,sx,y,ax);
    atom ax ♯ (p,sx,y); atom y ♯ (p,sx); atom sx ♯ p⟧ ⟹
  Special_axP p = Ex v (Ex x (Ex ax (Ex y (Ex sx
                    (FormP (Var x) AND VarP (Var v) AND TermP (Var y) AND
                     AbstFormP (Var v) Zero (Var x) (Var ax) AND
                     SubstFormP (Var v) (Var y) (Var x) (Var sx) AND
                     p EQ Q_Imp (Var sx) (Q_Ex (Var ax)))))))"
```

This definition states that a specialisation axiom is created from a formula `x`, a variable `v` and a term `y`, combined by appropriate abstraction and substitution operations. Correctness means proving that this predicate exactly characterises the elements of the set `special_axioms`, which was used to define the HF calculus. The most complicated such scheme is the induction axiom HF3 (defined in Sect. 2.2), with its three quantifiers. The treatment of the induction axiom requires nearly 180 lines, of which 120 are devoted to proving correctness with respect to the HF calculus.

A proof of a theorem `y` is a sequence `s` of axioms and inference rules, ending with `y`:

```
definition Prf :: "hf ⇒ hf ⇒ hf ⇒ bool"
  where "Prf s k y ≡ BuildSeq (λx. x ∈ Axiom)
                              (λu v w. ModPon v w u ∨ Exists v u ∨ Subst v u) s k
```

Finally, `y` codes a theorem provided it has a proof:

```
definition Pf :: "hf ⇒ bool"
  where "Pf y ≡ (∃s k. Prf s k y)"
```

Having proved the correctness of the predicates formalising the axioms and rules, the correctness of `Pf` follows easily. (Świerczkowski's seven lines of proof start here.) One direction is proved by induction on the proof of `{} ⊢ α`.

```
lemma proved_imp_Pf: assumes "H ⊢ α" "H={}" shows "Pf ⟦⌜α⌝⟧e"
```

Here, we use the shadow predicates and work directly in Isabelle/HOL. The corresponding HF predicate, `PfP`, is (crucially) a Σ formula. Moreover, codes are ground terms. Therefore `PfP` $\ulcorner\alpha\urcorner$ is a Σ sentence and is formally provable. This is the main use of the theorem `Sigma_fm_imp_thm`.

```
corollary proved_imp_proved_PfP: "{} ⊢ α ⟹ {} ⊢ PfP ⌜α⌝"
```

The reverse implication, despite its usefulness, is not always proved. Again using the rule `BuildSeq_induct`, it holds by induction on the length of the coded proof of $\ulcorner\alpha\urcorner$. The groundwork for this result involves proving, for each coded axiom and inference rule, that there exists a corresponding proof step in the HF calculus. We continue to work at the level of shadow predicates.

```
lemma Prf_imp_proved: assumes "Prf s k x" shows "∃A. x = ⟦⌜A⌝⟧e ∧ {} ⊢ A"
```

The corresponding result for `Pf` is immediate:

```
corollary Pf_quot_imp_is_proved: "Pf ⟦⌜α⌝⟧e ⟹ {} ⊢ α"
```

Now `{} ⊢ PfP` $\ulcorner\alpha\urcorner$ implies `Pf` $\llbracket\ulcorner\alpha\urcorner\rrbracket$e simply by the soundness of the calculus. It is now easy to show that the predicate `PfP` corresponds exactly to deduction in the HF calculus. Świerczkowski calls this result the *proof formalisation condition*.

```
theorem proved_iff_proved_PfP: "{} ⊢ α ⟷ {} ⊢ PfP ⌜α⌝"
```

Remark: `PfP` includes an additional primitive inference, substitution:

$$\frac{H \vdash \alpha}{H \vdash \alpha(t/x)}$$

This inference is derivable in the HF calculus, but the second incompleteness theorem requires performing substitution inferences, and reconstructing the derivation of substitution within `PfP` would be infeasible. Including substitution in the definition of `PfP` makes such steps immediate without complicating other proofs. Świerczkowski avoids this complication: his HF calculus [32] includes a substitution rule alongside a simpler specialisation axiom.

4.5 Pseudo-Functions

The HF calculus contains no function symbols other than $\lhd$. All other "functions" must be declared as predicates, which are mere abbreviations of formulas. This abuse of notation is understood in a standard way. The formula $\phi(f(x))$ can be taken as an abbreviation for $\exists y\,[F(x,y) \wedge \phi(y)]$ where F is the relation representing the function f and provided that F can be proved to be single valued: $F(x,y), F(x,y') \vdash y' = y$. Then f is called a *pseudo-function* and something like $f(x)$ is called a *pseudo-term*. Pseudo-terms do not actually exist, which will cause problems later.

Gödel formalised all syntactic operations as primitive recursive functions, while Boolos [2] used Δ formulas. With both approaches, much effort is necessary to admit a function definition in the first place. But then, it is known to be a function. Here we see a drawback of Świerczkowski's decision to base the formalisation on the notion of Σ formulas: they do not cover the property of being single valued. A predicate that corresponds to a function must be proved to be single valued within the HF calculus itself. Gödel's proof uses substitution as a function. The proof that substitution (on terms and formulas) is single valued requires nearly 500 lines in Isabelle/HOL, not counting considerable preparatory material (such as the partial ordering properties of $<$) mentioned in Sect. 3.2 above.

Fortunately, these proofs are conceptually simple and highly repetitious, and again much of the proof development can be done by cut and paste. The first step is to prove an *unfolding lemma* about the sequence buildup: if the predicate holds, then either the base case holds, or else there exist values earlier in the sequence for which one of the recursive cases can be applied. The single valued theorem is proved by complete induction on the length of the sequence, with a fully quantified induction formula (analogous to $\forall xyy'\,[F(x,y) \to F(x,y') \to y' = y]$) so that the induction hypothesis says that all shorter sequences are single valued for all possible arguments. All that is left is to apply the unfolding lemma to both instances of the relation F, and then to consider all combinations of cases. Most will be trivially contradictory, and in those few cases where the result has the same outer form, an appeal to the induction hypothesis for the operands will complete the proof.

Overall, the Gödel development proves single valued theorems for 12 predicates. Five of the theorems are proved by induction as sketched above. Here is an example:

```
lemma SeqSubstFormP_unique:
  "{SeqSubstFormP v a x y s u, SeqSubstFormP v a x y' s' u'} ⊢ y' EQ y"
```

The remaining results are straightforward corollaries of those inductions:

```
theorem SubstFormP_unique:
  "{SubstFormP v tm x y, SubstFormP v tm x y'} ⊢ y' EQ y"
```

It is worth repeating that these proofs are formally conducted within the HF calculus, essentially by single-step inferences. Meta-theory is no help here.

5 Gödel's First Incompleteness Theorem

Discussions involving encodings frequently need a way to refer to syntactic objects. We often see the convention where if x is a natural number, then a boldface $\boldsymbol{x}$ stands for the corresponding numeral. Then in expressions like $x = y \to \mathrm{Pf}\ulcorner \boldsymbol{x} = \boldsymbol{y}\urcorner$, we see that the boldface convention actually abbreviates the function $x \mapsto \boldsymbol{x}$, which needs to be formalisable in the HF calculus.

Thus, we need to define a function Q such that $Q(x) = \ulcorner x \urcorner$, in other words, $Q(x)$ yields some term t whose denotation is x. This is trivial if x ranges over the natural numbers, by primitive recursion. It is problematical when x ranges over sets, because it requires a canonical ordering over the universe of sets. We don't need to solve this problem just yet: the *first* incompleteness theorem needs only a function H such that $H(\ulcorner A \urcorner) = \ulcorner\ulcorner A \urcorner\urcorner$ for all A. Possible arguments of H are not arbitrary sets, but only nested tuples of ordinals; these have a canonical form, so a functional relationship is easy to define [32]. A certain amount of effort establishes the required property:[6]

```
lemma prove_HRP: fixes A::fm shows "{} ⊢ HRP ⌈A⌉ ⌈⌈A⌉⌉"
```

Note that the function H has been formalised as the relation `HRP`; it is defined using sequence operators, `LstSeqP`, etc., as we have seen already.

In order to prove Gödel's diagonal lemma, we need a function K_i to substitute the code of a formula into itself, replacing the variable x_i. This function, again, is realised as a relation, combining `HRP` with `SubstFormP`.

```
nominal_primrec KRP :: "tm ⇒ tm ⇒ tm ⇒ fm"
  where "atom y ♯ (v,x,x') ⟹
        KRP v x x' = Ex y (HRP x (Var y) AND SubstFormP v (Var y) x x')"
```

We easily obtain a key result: $K_i\,(\ulcorner \alpha \urcorner) = \ulcorner \alpha(i := \ulcorner \alpha \urcorner)\urcorner$.

```
lemma prove_KRP: "{} ⊢ KRP ⌈Var i⌉ ⌈α⌉ ⌈α(i::=⌈α⌉)⌉"
```

However, it is essential to prove that `KRP` behaves like a function. The predicates `KRP` and `HRP` can be proved to be single valued using the techniques discussed in the previous section. Then an appeal to `prove_KRP` uniquely characterises K_i as a function:

```
lemma KRP_subst_fm: "{KRP ⌈Var i⌉ ⌈α⌉ (Var j)} ⊢ Var j EQ ⌈α(i::=⌈α⌉)⌉"
```

Twenty five lines of tricky reasoning are needed to reach the next milestone: the *diagonal lemma*. Świerczkowski writes

> We replace the variable x_i in α by the [pseudo-term $K_i(x_i)$], and we denote by β the resulting formula. [32, p. 22]

The elimination of the pseudo-function K_i in favour of an existential quantifier involving `KRP` yields the following not-entirely-obvious Isabelle definition:

[6] Here **fixes** `A::fm` declares `A` to be a formula in the overloaded notation $\lceil A \rceil$. Świerczkowski uses α, β, ... to denote formulas, but I've frequently used the traditional A, B, ...

```
theorem Goedel_I:
  assumes "¬ {} ⊢ Fls"
  obtains δ where "{} ⊢ δ IFF Neg (PfP ⌈δ⌉)"  "¬ {} ⊢ δ"  "¬ {} ⊢ Neg δ"
                  "eval_fm e δ"  "ground_fm δ"
proof -
  fix i::name
  obtain δ where        "{} ⊢ δ IFF Neg ((PfP (Var i))(i::=⌈δ⌉))"
            and suppd:  "supp δ = supp (Neg (PfP (Var i))) - {atom i}"
    by (metis SyntaxN.Neg diagonal)
  then have diag: "{} ⊢ δ IFF Neg (PfP ⌈δ⌉)"
    by simp
  then have np: "¬ {} ⊢ δ ∧ ¬ {} ⊢ Neg δ"
    by (metis Iff_MP_same NegNeg_D Neg_D Neg_cong assms proved_iff_proved_PfP)
  then have "eval_fm e δ" using hfthm_sound [where e=e, OF diag]
    by simp (metis Pf_quot_imp_is_proved)
  moreover have "ground_fm δ" using suppd
    by (simp add: supp_conv_fresh ground_fm_aux_def subset_eq)
       (metis fresh_ineq_at_base)
  ultimately show ?thesis
    by (metis diag np that)
qed
```

Fig. 1 Gödel's First Incompleteness Theorem

```
  def β ≡ "Ex j (KRP ⌈Var i⌉ (Var i) (Var j) AND α(i ::= Var j))"
```

Note that one occurrence of `Var i` is quoted and the other is not. The development is full of pitfalls such as these.

The statement of the diagonal lemma is as follows. The second assertion states that the free variables of δ, the diagonal formula, are those of α, the original formula, minus `i`.

```
lemma diagonal:
  obtains δ where "{} ⊢ δ IFF α(i::=⌈δ⌉)"  "supp δ = supp α - {atom i}"
```

Figure 1 presents the proof of the first incompleteness theorem itself. The top level argument is quite simple, given the diagonal lemma. The key steps of the proof should be visible even to somebody who is not an Isabelle expert, thanks to the structured Isar language. Note that if `{} ⊢ Neg δ`, then `{} ⊢ PfP ⌈δ⌉` and therefore `{} ⊢ δ` by the proof formalisation condition, violating the assumption of consistency.

6 Towards the Second Theorem: Pseudo-Coding and Quotations

The second incompleteness theorem [1] has always been more mysterious than the first. Gödel's original paper [8] was designated "Part I" in anticipation of a subsequent "Part II" proving the second theorem, but no second paper appeared. Logicians recognised that the second theorem followed from the first, assuming that the first could itself be formalised in the internal calculus. While this assumption seems to be widely accepted, conducting such a formalisation explicitly remains difficult, even given today's theorem-proving technology.

A simpler route to the theorem involves the *ert-Bernays derivability conditions* [2, p. 15][9, p. 73].

$$\text{If } \vdash \alpha, \text{ then } \vdash \mathrm{Pf}(\ulcorner\alpha\urcorner) \tag{D1}$$

$$\text{If } \vdash \mathrm{Pf}(\ulcorner\alpha \to \beta\urcorner) \text{ and } \vdash \mathrm{Pf}(\ulcorner\alpha\urcorner), \text{ then } \vdash \mathrm{Pf}(\ulcorner\beta\urcorner) \tag{D2}$$

$$\text{If } \alpha \text{ is a strict } \Sigma \text{ sentence, then } \vdash \alpha \to \mathrm{Pf}(\ulcorner\alpha\urcorner) \tag{D3}$$

(Where there is no ambiguity, we identify Pf with the formalised predicate `PfP`; the latter is the actual HF predicate, but the notation Pf is widely used in the literature, along with Gödel's original Bew.)

Condition (D1) is none other than the theorem `proved_iff_proved_PfP` mentioned in Sect. 4.4 above. Condition (D2) seems clear by the definition of the predicate Pf, although all details of the workings of this predicate must be proved using low-level inferences in the HF calculus. Condition (D3) can be regarded as a version of the theorem `Sigma_fm_imp_thm` ("true Σ sentences are theorems") internalised as a theorem of the internal calculus. So while we avoid having to formalise the whole of Gödel's theorem within the calculus, we end up formalising a key part of it.

Condition (D3) is stated in a general form, but we only need one specific instance:

$$\vdash \mathrm{Pf}(\ulcorner\alpha\urcorner) \to \mathrm{Pf}(\ulcorner\mathrm{Pf}(\ulcorner\alpha\urcorner)\urcorner).$$

Despite a superficial resemblance, (D3) does not follow from (D1), which holds by induction on the proof of $\vdash \alpha$. As Świerczkowski explains [32, p. 23], condition (D3) is not general enough to prove by induction. In the sequel, we generalise and prove it.

6.1 Pseudo-Coding

Condition (D3) can be proved by induction on α if the assertion is generalised so that α can have free variables, say $x_1, \ldots, x_n$:

$$\vdash \alpha(x_1, \ldots, x_n) \to \mathrm{Pf}(\ulcorner\alpha(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})\urcorner)$$

The syntactic constructions used in this formula have to be formalised, and the necessary transformations have to be justified within the HF calculus. As mentioned above (Sect. 5), the boldface convention needs to be made rigorous. In particular, codings are always ground HF terms, and yet $\ulcorner\alpha(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})\urcorner$ has a functional dependence (as an HF term) on $x_1, \ldots, x_n$.

The first step in this process is to generalise coding to allow certain variables to be preserved as variables in the coded term. Recall that with normal quotations, every occurrence of a variable is replaced by the code of the variable name, ultimately a positive integer:[7]

```
function quot_dbtm :: "dbtm ⇒ tm"
  where
   "quot_dbtm DBZero = Zero"
 | "quot_dbtm (DBVar name) = ORD_OF (Suc (nat_of_name name))"
 | ...
```

Now let us define the *V-code* of a term or formula, where V is a set of variables to be preserved in the code:

[7] `ORD_OF (Suc n)` denotes an HF term that denotes a positive integer, even if `n` is a variable.

```
function vquot_dbtm :: "name set ⇒ dbtm ⇒ tm"
  where
   "vquot_dbtm V DBZero = Zero"
 | "vquot_dbtm V (DBVar name) = (if name ∈ V then Var name
                                 else ORD_OF (Suc (nat_of_name name)))"
 | ...
```

V-coding is otherwise identical to standard coding, with the overloaded syntax `⌊A⌋V`. The parameter V is necessary because not all variables should be preserved; it will be necessary to consider a series of V-codes for $V = \emptyset$, $\{x_1\}$ ..., $\{x_1, \ldots, x_n\}$.

6.2 Simultaneous Substitution

In order to formalise the notation $\ulcorner \alpha(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) \urcorner$, it is convenient to introduce a function for simultaneous substitution. Here Świerczkowski's presentation is a little hard to follow:

> Suppose β is a theorem, i.e., $\vdash \beta$. If we replace each of the variables at each of its free occurrences in β by some constant term then the formula so obtained is also a theorem (by the Substitution Rule...). This just described situation in the meta-theory admits description in HF. [32, p. 24]

It took me weeks of failed attempts to grasp the meaning of the phrase "constant term". It does not mean a term containing no variables, but a term satisfying the predicate `ConstP` and thus denoting the *code* of a constant. Formalising this process seems to require replacing each variable x_i by a new variable, x_i', intended to denote $\boldsymbol{x_i}$. Later, it will be constrained to do so by a suitable HF predicate. And so we need a function to perform simultaneous substitutions in a term for all variables in a set V. Using a "fold" operator over finite sets [19] eliminates the need to consider the variables in any particular order.

```
definition ssubst :: "tm ⇒ name set ⇒ (name ⇒ tm) ⇒ tm"
  where "ssubst t V F = Finite_Set.fold (λi. subst i (F i)) t V"
```

The renaming of x_i to x_i' could be done using arithmetic on variable subscripts, but the formalisation instead follows an abstract approach, using nominal primitives. An Isabelle locale defines a proof context containing a permutation `p` (mapping all variable names to new ones), a finite set `Vs` of variable names and finally the actual renaming function `F`, which simply applies the permutation to any variable in `Vs`.[8]

```
locale quote_perm =
  fixes p :: perm and Vs :: "name set" and F :: "name ⇒ tm"
  assumes p: "atom ‘ (p · Vs) ♯* Vs"
      and pinv: "-p = p"
      and Vs:  "finite Vs"
  defines "F ≡ make_F Vs p"
```

Most proofs about `ssubst` are done within the context of this locale, because it provides sufficient conditions for the simultaneous substitution to be meaningful. The first condition states that `p` maps all the variables in `Vs` to variables outside of that set, while second condition states that `p` is its own inverse.

This abstract approach is a little unwieldy at times, but its benefits can be seen in the simple fact below, which states the effect of the simultaneous substitution on a single variable.

[8] `make_F Vs p i = Var (p · i)` provided `i ∈ Vs`.

```
lemma ssubst_Var_if:
  assumes "finite V"
    shows "ssubst (Var i) V F = (if i ∈ V then F i else Var i)"
```

We need to show that the variables in the set `Vs` can be renamed, one at a time, in a pseudo-coded de Bruijn term. Let `V ⊆ Vs` and suppose that the variables in `V` have already been renamed, and choose one of the remaining variables, `w`. It will be replaced by a new variable, computed by `F w`. And something very subtle is happening: the variable `w` is represented in the term by its code, `⌈Var w⌉`. Its replacement, `F w`, is `Var (p · w)` and a *variable*.

```
lemma SubstTermP_vquot_dbtm:
  assumes w: "w ∈ Vs - V" and V: "V ⊆ Vs" "V' = p · V"
      and s: "supp dbtm ⊆ atom ‘ Vs"
  shows
  "insert (ConstP (F w)) {ConstP (F i) | i. i ∈ V}
   ⊢ SubstTermP ⌈Var w⌉ (F w)
                (ssubst (vquot_dbtm V dbtm) V F)
                (subst w (F w) (ssubst (vquot_dbtm (insert w V) dbtm) V F))"
```

This theorem is proved by structural induction on `dbtm`, the de Bruijn term. The condition `supp dbtm ⊆ atom ‘ Vs` states that the free variables of `dbtm` all belong to `Vs`. The variable case of the induction is tricky (and is the crux of the entire proof). We are working with a coded term that contains both coded variables and real ones (of the form `F i`); it is necessary to show that the real variables are preserved by the substitution, because they are the $\boldsymbol{x_i}$ that we are trying to introduce. The `F i` are preserved under the assumption that they denote constants, which is the point of including the formulas `ConstP (F i)` for all `i ∈ V` on the left side of the turnstile. These assumptions will have to be justified later.

Under virtually the same assumptions (omitted), the analogous result holds for pseudo-coded de Bruijn formulas.

```
lemma SubstFormP_vquot_dbfm:
  "insert (ConstP (F w)) {ConstP (F i) | i. i ∈ V}
   ⊢ SubstFormP ⌈Var w⌉ (F w)
                (ssubst (vquot_dbfm V dbfm) V F)
                (subst w (F w) (ssubst (vquot_dbfm (insert w V) dbfm) V F))"
```

The proof is an easy structural induction on `dbfm`. Every case holds immediately by properties of substitution and the induction hypotheses or by the previous theorem, for terms. The only difficult case in these two proofs is the variable case discussed above. Using the notation for V-coding, we can see that the substitution predicate `SubstFormP` can transform the term `ssubst ⌊A⌋V V F` into

```
ssubst ⌊A⌋(insert w V) (insert w V) F.
```

Repeating this step, we can replace any finite set of variables in a coded formula by real ones, realising Świerczkowski's remark quoted at the top of this section, and in particular his last sentence. That is, if β is a theorem in HF (if $\vdash$ Pf β holds) then the result of substituting constants for its variables is also an HF theorem. More precisely still, we are replacing some subset `V` of the variables by fresh variables (the `F i`), constrained by the predicate `ConstP`.

```
theorem PfP_implies_PfP_ssubst:
  fixes β::fm
  assumes β: "{} ⊢ PfP ⌈β⌉"
      and V: "V ⊆ Vs"
      and s: "supp β ⊆ atom ‘ Vs"
    shows    "{ConstP (F i) | i. i ∈ V} ⊢ PfP (ssubst ⌊β⌋V V F)"
```

The effort needed to formalise the results outlined above is relatively modest, at 330 lines of Isabelle/HOL, but this excludes the effort needed to prove some essential lemmas, which state that the various syntactic predicates work correctly. Because these proofs concern non-ground HF formulas, theorem *Sigma_fm_imp_thm* does not help. Required is an HF formalisation of operations on sequences, such as concatenation. That in turn requires formalising further operations such as addition and set union. These proofs (which are conducted largely in the HF calculus) total over 2,800 lines. This total includes a library of results for truncating and concatenating sequences. Here is a selection of the results proved.

Substitution preserves the value *Zero*:

theorem *SubstTermP_Zero: "{TermP t} ⊢ SubstTermP ⌈Var v⌉ t Zero Zero"*

On terms constructed using *Eats* (recall that *Q_Eats* constructs the *code* of an *Eats* term), substitution performs the natural recursion.

```
theorem SubstTermP_Eats:
  "{SubstTermP v i t1 u1, SubstTermP v i t2 u2}
   ⊢ SubstTermP v i (Q_Eats t1 t2) (Q_Eats u1 u2)"
```

This seemingly obvious result takes nearly 150 lines to prove. The sequences for both substitution computations are combined to form a new sequence, which must be extended to yield the claimed result and shown to be properly constructed.

Substitution preserves constants. This fact is proved by induction on the sequence buildup of the constant *c*, using the previous two facts about *SubstTermP*.

theorem *SubstTermP_Const: "{ConstP c, TermP t} ⊢ SubstTermP ⌈Var w⌉ t c c"*

Each recursive case of a syntactic predicate must be verified using the techniques outlined above for *SubstTermP_Eats*. Even when there is only a single operand, as in the following case for negation, the proof is around 100 lines.

theorem *SubstFormP_Neg: "{SubstFormP v i x y} ⊢ SubstFormP v i (Q_Neg x) (Q_Neg y)"*

A complication is that *LstSeqP* accepts sequences that are longer than necessary, and these must be truncated to a given length before they can be extended. These lengthy arguments must be repeated for each similar proof. So, for the third time, one of our chief tools is cut and paste.

Exactly the same sort of reasoning can be used to show that proofs can be combined as expected in order to apply inference rules. The following theorem expresses the Hilbert-Bernays derivability condition (D2):

theorem *PfP_implies_ModPon_PfP: "⟦H ⊢ PfP (Q_Imp x y); H ⊢ PfP x⟧ ⟹ H ⊢ PfP y"*

Now only one task remains: to show condition (D3).

6.3 Making Sense of Quoted Values

As mentioned in Sect. 5, making sense of expressions like $x = y \to \mathrm{Pf}\ \ulcorner \boldsymbol{x} = \boldsymbol{y} \urcorner$ requires a function Q such that $Q(x) = \ulcorner x \urcorner$:

$$Q(0) = \ulcorner 0 \urcorner = 0$$
$$Q(x \lhd y) = \langle \ulcorner \lhd \urcorner, Q(x), Q(y) \rangle$$

Trying to make this definition unambiguous, Świerczkowski [32] sketches a total ordering on sets, but the technical details are complicated and incomplete. The same ordering can be defined via the function $f : \mathrm{HF} \to \mathbb{N}$ such that $f(x) = \sum \{2^{f(y)} \mid y \in x\}$. It is intuitively clear, but formalising the required theory in HF would be laborious. It turns out that we do not need a canonical term $\boldsymbol{x}$ or a function Q. We only need a relation: *QuoteP* relates a set x to (the codes of) the terms that denote x.

The relation *QuoteP* can be defined using precisely the same methods as we have seen above for recursive functions, via a sequence buildup. The following facts can be proved using the methods described in the previous sections.

lemma *QuoteP_Zero: "{} ⊢ QuoteP Zero Zero"*

lemma *QuoteP_Eats:*
"{QuoteP t1 u1, QuoteP t2 u2} ⊢ QuoteP (Eats t1 t2) (Q_Eats u1 u2)"

It is also necessary to prove (by induction within the HF calculus) that for every x there exists some term $\boldsymbol{x}$.

lemma *exists_QuoteP:*
assumes *j: "atom j ♯ x"* **shows** *"{} ⊢ Ex j (QuoteP x (Var j))"*

We need similar results for all of the predicates involved in concatenating two sequences. They essentially prove that the corresponding pseudo-functions are total.

Now we need to start connecting these results with those of the previous section, which (following Świerczkowski) are proved for constants in general, although they are needed only for the outputs of *QuoteP*. An induction in HF on the sequence buildup proves that these outputs satisfy *ConstP*.

lemma *QuoteP_imp_ConstP: "{ QuoteP x y } ⊢ ConstP y"*

This is obvious, because *QuoteP* involves only *Zero* and *Q_Eats*, which construct quoted sets. Unfortunately, the proof requires the usual reasoning about sequences in order to show basic facts about *ConstP*, which takes hundreds of lines.

The main theorem from the previous section included the set of formulas

{ConstP (F i) | i. i ∈ V}

on the left of the turnstile, representing the assumption that all introduced variables denoted constants. Now we can replace this assumption by one expressing that the relation *QuoteP* holds between each pair of old and new variables.

definition *quote_all :: "[perm, name set] ⇒ fm set"*
where *"quote_all p V = {QuoteP (Var i) (Var (p · i)) | i. i ∈ V}"*

The relation *QuoteP (Var i) (Var (p · i)* holds between the variable *i* and the renamed variable *p · i*, for all *i ∈ V*. Recall that *p* is a permutation on variable names. By virtue of the theorem *QuoteP_imp_ConstP*, we obtain a key result, which will be used heavily in subsequent proofs for reasoning about coded formulas and the Pf predicate.

theorem *quote_all_PfP_ssubst:*
assumes *β: "{} ⊢ β"*
and *V: "V ⊆ Vs"*
and *s: "supp β ⊆ atom ‘ Vs"*
shows *"quote_all p V ⊢ PfP (ssubst ⌊β⌋V V F)"*

In English: Let $\vdash \beta$ be a theorem of HF whose free variables belong to the set *Vs*. Take the code of this theorem, $\lfloor\beta\rfloor$, and replace some subset *V ⊆ Vs* of its free variables by

new variables constrained by the `QuoteP` relation. The result, `ssubst ⌊β⌋V V F`, satisfies the provability predicate.

The reader of even a very careful presentation of Gödel's second incompleteness theorem, such as Grandy [9], will look in vain for a clear and rigorous treatment of the $\boldsymbol{x}$ (or $\underline{x}$) convention. Boolos [2] comes very close with his Bew[F] notation, but he is quite wrong to state "notice that Bew[F] has the same variables free as [the formula] F" [2, p. 45] when in fact they have no variables in common. Even Świerczkowski's highly detailed account is at best ambiguous. He consistently uses function notation, but his carefully-stated guidelines for replacing occurrences of pseudo-functions by quantified formulas [32, Sect. 5] are not relevant here. (This problem only became clear after a time-consuming attempt at a formalisation on that basis.) My companion paper [27], which is aimed at logicians, provides a more detailed discussion of these points. It concludes that these various notations obscure not only the formal details of the proof but also the very intuitions they are intended to highlight.

6.4 Proving $\vdash \alpha \to \mathrm{Pf}\,(\ulcorner \alpha \urcorner)$

We now have everything necessary to prove condition (D3):

$$\text{If } \alpha \text{ is a strict } \Sigma \text{ sentence, then } \vdash \alpha \to \mathrm{Pf}\,(\ulcorner \alpha \urcorner)$$

The proof will be by induction on the structure of α. As stated in Sect. 3.3 above, a strict Σ formula has the form $x \in y$, $\alpha \vee \beta$, $\alpha \wedge \beta$, $\exists x\, \alpha$ or $(\forall x \in y)\, \alpha$. Therefore, the induction will include one single base case,

$$x \in y \to \mathrm{Pf}\ \ulcorner \boldsymbol{x} \in \boldsymbol{y} \urcorner, \tag{3}$$

along with inductive steps for disjunction, conjunction, etc.

6.4.1 The Propositional Cases

The propositional cases are not difficult, but are worth examining as a warmup exercise. From the induction hypotheses $\vdash \alpha \to \mathrm{Pf}\,(\ulcorner \alpha \urcorner)$ and $\vdash \beta \to \mathrm{Pf}\,(\ulcorner \beta \urcorner)$, we must show

$$\vdash \alpha \vee \beta \to \mathrm{Pf}\,(\ulcorner \alpha \vee \beta \urcorner) \text{ and}$$
$$\vdash \alpha \wedge \beta \to \mathrm{Pf}\,(\ulcorner \alpha \wedge \beta \urcorner).$$

Both of these cases are trivial by propositional logic, given the lemmas $\vdash \mathrm{Pf}\,(\ulcorner \alpha \urcorner) \to \mathrm{Pf}\,(\ulcorner \alpha \vee \beta \urcorner)$, $\vdash \mathrm{Pf}\,(\ulcorner \beta \urcorner) \to \mathrm{Pf}\,(\ulcorner \alpha \vee \beta \urcorner)$ and

$$\vdash \mathrm{Pf}\,(\ulcorner \alpha \urcorner) \to \mathrm{Pf}\,(\ulcorner \beta \urcorner) \to \mathrm{Pf}\,(\ulcorner \alpha \wedge \beta \urcorner) \tag{4}$$

Proving (4) directly from the definitions would need colossal efforts, but there is a quick proof. The automation of the HF calculus is good enough to prove tautologies, and from $\vdash \alpha \to \beta \to \alpha \wedge \beta$, the proof formalisation condition[9] yields

$$\vdash \mathrm{Pf}\,(\ulcorner \alpha \to \beta \to \alpha \wedge \beta \urcorner)$$

Finally, the Hilbert-Bernays derivability condition (D2) yields the desired lemma, (4). This trick is needed whenever we need to do propositional reasoning with Pf .

[9] Of Sect. 4.4, but via the substitution theorem `quote_all_PfP_ssubst` proved above. The induction concerns generalised formulas involving pseudo-coding: `PfP (ssubst ⌊α⌋V V F)`.

6.4.2 The Equality and Membership Cases

Now comes one of the most critical parts of the formalisation. Many published proofs [2, 32] of the second completeness theorem use the following lemma:

$$x = y \to \mathrm{Pf}\ \ulcorner \boldsymbol{x} = \boldsymbol{y} \urcorner \tag{5}$$

This in turn is proved using a lemma stating that $x = y$ implies $\boldsymbol{x} = \boldsymbol{y}$, which is false here: we have defined `QuoteP` only as a relation, and even $\{0, 1\}$ can be written in two different ways. Nevertheless, the statement (5) is clearly true: if $\boldsymbol{x}$ and $\boldsymbol{y}$ are constant terms denoting x and y, respectively, where $x = y$, then $\mathrm{Pf}\ \ulcorner \boldsymbol{x} = \boldsymbol{y} \urcorner$ holds. The equivalence of two different ways of writing a finite set should obviously be provable. This problem recalls the situation described in 3.3 above, and the induction used to prove `Subset_Mem_sf_lemma`. The solution, once again, is to prove the conjunction

$$(x \in y \to \mathrm{Pf}\ \ulcorner \boldsymbol{x} \in \boldsymbol{y} \urcorner) \wedge (x \subseteq y \to \mathrm{Pf}\ \ulcorner \boldsymbol{x} \subseteq \boldsymbol{y} \urcorner)$$

by induction (in HF) on the sum of the sizes of $\boldsymbol{x}$ and $\boldsymbol{y}$. The proof is huge (nearly 340 lines), with eight universal quantifiers in the induction formula, each of which must be individually instantiated in order to apply an induction hypothesis.

```
⊢ All i (All i' (All si (All li (All j (All j' (All sj (All lj
   (SeqQuoteP (Var i) (Var i') (Var si) (Var li) IMP
    SeqQuoteP (Var j) (Var j') (Var sj) (Var lj) IMP
    HaddP (Var li) (Var lj) (Var k) IMP
      ( (Var i IN Var j IMP PfP (Q_Mem (Var i') (Var j')))  AND
        (Var i SUBS Var j IMP PfP (Q_Subset (Var i') (Var j'))))))))))))"
```

Using `SeqQuoteP` (which describes the sequence computation of `QuoteP`) gives access to a size measure for the two terms, which are here designated `i` and `j`. Their sizes, `li` and `lj`, are added using `HaddP`, which is simply addition as defined in the HF calculus. (This formalisation of addition is also needed for reasoning about sequences.) Their sum, `k`, is used as the induction variable.

Although the second half of the conjunction suffices to prove (5), it is never needed outside of the induction, and neither is (5) itself. All we need is (3). And so we reach the next milestone.

```
lemma
  assumes "atom i ♯ (j,j',i')" "atom i' ♯ (j,j')" "atom j ♯ (j')"
  shows QuoteP_Mem_imp_QMem:
        "{QuoteP (Var i) (Var i'), QuoteP (Var j) (Var j'), Var i IN Var j}
         ⊢ PfP (Q_Mem (Var i') (Var j'))"
    and QuoteP_Mem_imp_QSubset:
        "{QuoteP (Var i) (Var i'), QuoteP (Var j) (Var j'), Var i SUBS Var j}
         ⊢ PfP (Q_Subset (Var i') (Var j'))"
```

Turning to the main induction on α, the notoriously "delicate" [2, p. 48] case is bounded universal quantification. Many of the delicate points here are connected with the way the nominal approach is used. We need to maintain and extend a permutation relating old and new variable names. Such complexities are evident in mathematical texts, in their treatment of variable names [32, Lemma 9.7].

```
lemma (in quote_perm) quote_all_Mem_imp_All2:
  assumes IH: "insert (QuoteP (Var i) (Var i')) (quote_all p Vs)
               ⊢ α IMP PfP (ssubst ⌊α⌋(insert i Vs) (insert i Vs) Fi)"
      and "supp (All2 i (Var j) α) ⊆ atom ‘ Vs"
```

```
    and j: "atom j ♯ (i,α)" and i: "atom i ♯ p" and i': "atom i' ♯ (i,p,α)"
  shows "insert (All2 i (Var j) α) (quote_all p Vs)
        ⊢ PfP (ssubst ⌊All2 i (Var j) α⌋Vs Vs F)"
```

The final case, for the existential quantifier, is also somewhat complicated to formalise. The details are again mostly connected with managing free and bound variable names using nominal methods, and are therefore omitted. We can conclude our discussion of the inductive argument by viewing the final result:

```
lemma star:
  assumes "ss_fm α"  "finite V"  "supp α ⊆ atom ‘ V"
    shows "insert α (quote_all p V) ⊢ PfP (ssubst ⌊α⌋V V F)"
```

Condition (D3) now follows easily, since the formula α is then a sentence. Although some technical conditions (involving variable names and permutations) have been omitted from the previous two theorems, our main result below appears exactly as it was proved. Of course, $\alpha \vdash \mathrm{Pf}\ulcorner\alpha\urcorner$ is equivalent to $\vdash \alpha \to \mathrm{Pf}\ulcorner\alpha\urcorner$.

```
theorem Provability:
  assumes "Sigma_fm α" "ground_fm α"
    shows "{α} ⊢ PfP ⌈α⌉"
```

7 Gödel's Second Incompleteness Theorem

The final steps of the second incompleteness theorem can be seen in Fig. 2. The diagonal formula, δ, is obtained via the first incompleteness theorem. Then we can quickly establish both $\mathrm{Pf}\ulcorner\delta\urcorner \vdash \mathrm{Pf}(\ulcorner\mathrm{Pf}\ulcorner\delta\urcorner\urcorner)$ and $\mathrm{Pf}\ulcorner\delta\urcorner \vdash \mathrm{Pf}(\ulcorner\neg\mathrm{Pf}\ulcorner\delta\urcorner\urcorner)$. These together imply $\mathrm{Pf}\ulcorner\delta\urcorner \vdash \mathrm{Pf}(\ulcorner\bot\urcorner)$ using a variant of condition (D2). It follows that if the system proves its own consistency, then it also proves $\vdash \neg\mathrm{Pf}\ulcorner\delta\urcorner$ and therefore $\vdash \delta$, a contradiction.

Świerczkowski [32] presents a few other results which have not been formalised here. These include a refinement of the incompleteness theorem (credited to Reinhardt) and a theory of a linear order on the HF sets, but recall that claim (5) can be proved without using any such ordering. The approach adopted here undoubtedly involves less effort than formalising the ordering in the HF calculus.

The total proof length of nearly 12,400 lines comprises around 4,600 lines for the second theorem and 7,700 lines for the first.[10] (One could also include 2,700 lines for HF set theory itself, but we would not count the standard libraries of natural numbers if they were used as the basis of the proof.) O'Connor's proof comprises 47,000 lines of Coq, while Shankar's takes 20,000 lines [30, p. 139] and Harrison's [10] is a miniscule 4,400 lines of HOL Light. Recall that none of these other proofs include the second incompleteness theorem.

But comparisons are almost meaningless because of the enormous differences among the formalisations. Shankar wrote (and proved to be representable) a LISP interpreter for coding up the metatheory [30]; this was a huge effort, but then the various coding functions could then be written in LISP without further justification. He also used HF for coding, presumably because of its similarity to LISP S-expressions. O'Connor formalised a very general syntax for first-order logic [22]. He introduced a general inductive definition of the primitive recursive functions, but proving specific functions to be primitive recursive turned out to be extremely difficult [23, Sect. 5.3]. Harrison has

[10] Prior to polishing and removing unused material, the proof totalled 17,000 lines.

```
theorem Goedel_II:
  assumes "¬ {} ⊢ Fls"
    shows "¬ {} ⊢ Neg (PfP ⌈Fls⌉)"
proof -
  from assms Goedel_I obtain δ
    where diag: "{} ⊢ δ IFF Neg (PfP ⌈δ⌉)"  "¬ {} ⊢ δ"
      and gnd:  "ground_fm δ"
    by metis
  have "{PfP ⌈δ⌉} ⊢ PfP ⌈PfP ⌈δ⌉⌉"
    by (auto simp: Provability ground_fm_aux_def supp_conv_fresh)
  moreover have "{PfP ⌈δ⌉} ⊢ PfP ⌈Neg (PfP ⌈δ⌉)⌉"
    apply (rule MonPon_PfP_implies_PfP [OF _ gnd])
    apply (metis Conj_E2 Iff_def Iff_sym diag(1))
    apply (auto simp: ground_fm_aux_def supp_conv_fresh)
    done
  moreover have "ground_fm (PfP ⌈δ⌉)"
    by (auto simp: ground_fm_aux_def supp_conv_fresh)
  ultimately have "{PfP ⌈δ⌉} ⊢ PfP ⌈Fls⌉" using PfP_quot_contra
    by (metis (no_types) anti_deduction cut2)
  thus "¬ {} ⊢ Neg (PfP ⌈Fls⌉)"
    by (metis Iff_MP2_same Neg_mono cut1 diag)
qed
```

Fig. 2 Gödel's Second Incompleteness Theorem

not published a paper describing his formalisation, but devotes a few pages to Gödel's theorems in his *Handbook of Practical Logic* [12, p. 546–555], including extracts of HOL Light proofs. He defines Σ_1 and Π_1 formulas and quotes some meta-theoretical results relating truth and provability.

This project took approximately one year, in time left available after fulfilling teaching and administrative duties. The underlying set theory took only two weeks to formalise. The Gödel development up to the proof formalisation condition took another five months. From there to the first incompleteness theorem took a further two months, mostly devoted to proving single valued properties. Then the second incompleteness theorem took a further four months, including much time wasted due to misunderstanding this perplexing material. Some material has since been consolidated or streamlined. The final version is available online [26].

7.1 The Lengths of Proofs

Figure 3 depicts the sizes of the Isabelle/HOL theories making up various sections of the proof development. The first theorem takes up the bulk of the effort. Apart from the massive HF proofs about predicates, which are mostly of obvious properties, the second theorem appears to be a fairly easy step given the first. Why then has it not been formalised until now? A reasonable guess is that previous researchers were not aware of Świerczkowski's [32] elaborate development. The most detailed of the previous proofs [2, 9] left too much to the imagination, and even Świerczkowski makes some errors. He devotes much of his Sect. 7 to proofs concerning substitution for (non-existent) pseudo-terms analogous to $\boldsymbol{x}$. Recall that pseudo-terms are merely a notational shorthand to allow function syntax, and are not actual terms. Finally, there was the critical issue of the ordering on the HF sets. Solving these mysteries required much thought, and the

first completed formalisation included thousands of lines of proofs belonging to aborted attempts.

A discussion of the de Bruijn coefficient (the expansion in size entailed by the process of formalisation) is always interesting, but difficult to do rigorously. In this case, the formalisation required proving a great many theorems that were not even hinted at in the source text, for example, setting up a usable sequent calculus (Świerczkowski merely gives the rudiments of a Hilbert system), or proving that the various codings of syntax actually work (virtually all authors take this for granted), or proving that "functions" are functions. The hundreds of lines of single-step HF calculus proofs are the single largest contributor to the size of this development, and such things never appear in standard presentations of the incompleteness theorems.

A crude calculation yields 30 pages at 35 lines per page or 1050 lines for Świerczkowski's proof, compared with 12,400 lines of Isabelle, for a de Bruijn factor of 12. Focusing on just the proof of the first incompleteness theorem (after the preliminary developments), we have about 80 lines of informal text and 671 lines of Isabelle, giving a factor of 8.4; that includes 150 lines in the Isabelle script to prove that functions are single valued.

A further issue is the heavy use of cut and paste in the HF calculus proofs. Better automation for HF could help, but spending time to develop a tactic that will only be called a few times is hard to justify. An alternative idea is to define higher-order operators for the sequence constructions, which could be proved to be functional once and for all. However, higher-order operators are difficult to define using nominal syntax. Perhaps it could be attempted using naive variables.

[scale=0.6]theory-sizes

Fig. 3 Sizes of Constituent Theories

7.2 The Formalisation of Variable Binding

The role of bound variable syntax remains unclear. Shankar [30, 31] used de Bruijn variables to formalise the Church-Rosser theorem but not the incompleteness theorem. Harrison did not use them either. O'Connor also used traditional syntactic bound variables, but complained about huge complications concerning substitution (recall Sect. 2.3). The present development uses the nominal package, easing many proofs, but at a price: over 900 lines involve freshness specifications, and around 70 lemmas involving freshness are proved. Moreover, many proof steps are slow. While the project was underway, proof times taking minutes were not unusual. Even after recent improvements to the nominal package, they can take tens of seconds. Additional performance improvements would be welcome, as well as a more concise notation for freshness conditions when many new names are needed.

In fairness to the nominal approach, explicit variable names would also have to be fresh and analogous assertions would be necessary, along with some procedure for calculating new names numerically and proving them to be fresh. The effort may be similar either way, but the nominal approach is more abstract and natural: who after all refers to specific, calculated variable names in textbook proofs?

My first attempt to formalise the incompleteness theorems used explicit names. Then, substitution on formulas was only available as a relation, and many proofs required numerical operations on variable names. This effort would have multiplied considerably for the second incompleteness theorem. Using de Bruijn indices for HF syntax was not attractive: I had previously formalised Gödel's definition of the constructible sets and his proof of the relative consistency of the axiom of choice [25]. Here it was also necessary to define a great many predicates within an encoding of first-order logic. This work was done in Isabelle/ZF, a version of Isabelle for axiomatic set theory. I used de Bruijn indices in these definitions, but the loss of readability was a severe impediment to progress.

It is worth investigating how this formalisation would be affected by the change to another treatment of variable binding. As regards the Gödel numbering of formulas, the use of de Bruijn variables can be called an unqualified success. It was easy to set up and all necessary properties were proved without great difficulties.

7.3 On Verifying Proof Assistants

In a paper entitled "Towards Self-verification of HOL Light", Harrison says,

> Gödel's second incompleteness theorem tells us that [a logical system] cannot prove its own consistency in any way at all .... So, regardless of implementation details, if we want to prove the consistency of a proof checker, we need to use a logic that in at least some respects goes beyond the logic the checker itself supports. [11, p. 179]

This statement is potentially misleading, and has given rise to the mistaken view that it is impossible to verify a proof checker in its own logic.

Harrison's aim is to prove that HOL Light cannot prove the theorem `FALSE`, and this indeed requires proving the consistency of higher-order logic itself. Unfortunately, most consistency proofs are unsatisfactory because they more or less assume the desired conclusion: they are thinly disguised versions of the tautology $\mathrm{Con}(L) \conj L \imp \mathrm{Con}(L)$. This is a consequence of the second incompleteness theorem, since the consistency of L can only be proved in a strictly stronger formal system.

Mathematicians accept strong formal systems, such as ZF set theory, with little justification other than intuition and experience. Moreover, they examine very strong further axioms. The axiom of constructibility is an instructive case: it is known to be relatively consistent with respect to the axioms of set theory, but it is not generally accepted as true [16, p. 170]. The standard ZF axioms are generally regarded as true, although they cannot even be proved to be consistent. Thus we have no good way of proving consistency, and yet consistency does not guarantee truth.

This situation calls for a separation of concerns. The builders of verification tools should be concerned with the correctness of their code, but the correctness of the underlying formal calculus is the concern of logicians. Harrison notes that "almost all implementation bugs in HOL Light and other versions of HOL have involved variable renaming" [11, p. 179], and this type of issue should be our focus. Verifying a proof assistant involves verifying that it implements a data structure for the assertions of the formal calculus and that it satisfies a commuting diagram relating deductions on the implemented assertions with the corresponding deductions in the calculus. Gödel's theorems have no relevance here.

8 Conclusions

The main finding is simply that Gödel's second incompleteness theorem can be proved with a relatively modest effort, in only a few months starting with a proof of the first incompleteness theorem. While the nominal approach to syntax is clearly not indispensable, it copes convincingly with a development of this size and complexity. The use of HF set theory as an alternative to Peano arithmetic is clearly justified, eliminating the need to formalise basic number theory within the embedded calculus; the necessary effort to do that would greatly exceed the difficulties (mentioned in Sect. 6.4 above) caused by the lack of a simple canonical ordering on HF sets.

Many published proofs of the incompleteness theorems replace technical proofs by vague appeals to Church's thesis. Boolos [2] presents a more detailed and careful exposition, but still leaves substantial gaps. Even the source text [32] for this project, although written with great care, had problems: a significant gap (concerning the canonical ordering of HF sets), a few minor ones (concerning Σ formulas, for example), and pages of material that are, at the very least, misleading. These remarks are not intended as criticism but as objective observations of the complexity of this material, with its codings of codings. A complete formal proof, written in a fairly readable notation, should greatly clarify the issues involved in these crucially important theorems.

Acknowledgment

Jesse Alama drew my attention to Świerczkowski [32], the source material for this project. Christian Urban assisted with nominal aspects of some of the proofs, even writing code. Brian Huffman provided the core formalisation of type `hf`. Dana Scott offered advice and drew my attention to Kirby [15]. Matt Kaufmann and the referees made many insightful comments.

References

1. Joan Bagaria. A short guide to Gödel's second incompleteness theorem. *Teorema*, 22(3):5–15, 2003.
2. George Stephen Boolos. *The Logic of Provability*. Cambridge University Press, 1993.
3. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
4. S. Feferman, editor. *Kurt Gödel: Collected Works*, volume I. Oxford University Press, 1986.
5. Torkel Franzén. *Gödel's Theorem: An Incomplete Guide to Its Use and Abuse*. A K Peters, 2005.
6. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
7. Kurt Gödel. On completeness and consistency. In Feferman [4], pages 234–236.
8. Kurt Gödel. On formally undecidable propositions of Principia Mathematica and related systems. In Feferman [4], pages 144–195. First published in 1931 in the *Monatshefte für Mathematik und Physik*.
9. Richard E Grandy. *Advanced Logic for Applications*. Reidel, 1977.
10. John Harrison. Re: Re: Gödel's incompleteness theorem. Email dated 15 January 2014.
11. John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning — Third International Joint Conference, IJCAR 2006*, LNAI 4130, pages 177–191. Springer, 2006.

12. John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
13. Richard E Hodel. *An Introduction to Mathematical Logic*. PWS Publishing Company, 1995.
14. Joe Hurd and Tom Melham, editors. *Theorem Proving in Higher Order Logics: TPHOLs 2005*, LNCS 3603. Springer, 2005.
15. Laurence Kirby. Addition and multiplication of sets. *Mathematical Logic Quarterly*, 53(1):52–65, 2007.
16. Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*. North-Holland, 1980.
17. Andreas Lochbihler. Formalising finfuns — generating code for functions as data from Isabelle/HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 310–326. Springer, 2009.
18. Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.
19. Tobias Nipkow and Lawrence C. Paulson. Proof pearl: Defining functions over finite sets. In Hurd and Melham [14], pages 385–396.
20. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. An up-to-date version is distributed with Isabelle.
21. Michael Norrish and René Vestergaard. Proof pearl: de Bruijn terms really do work. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2007*, LNCS 4732, pages 207–222. Springer, 2007.
22. Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In Hurd and Melham [14], pages 245–260.
23. Russell S. S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. PhD thesis, Radboud University Nijmegen, 2009.
24. Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.
25. Lawrence C. Paulson. The relative consistency of the axiom of choice — mechanized using Isabelle/ZF. *LMS Journal of Computation and Mathematics*, 6:198–248, 2003. `http://www.lms.ac.uk/jcm/6/lms2003-001/`.
26. Lawrence C. Paulson. Gödel's incompleteness theorems. *Archive of Formal Proofs*, November 2013. `http://afp.sf.net/entries/Incompleteness.shtml`, Formal proof development.
27. Lawrence C. Paulson. A machine-assisted proof of Gödel's incompleteness theorems for the theory of hereditarily finite sets. *Review of Symbolic Logic*, 7(3):484–498, September 2014.
28. Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.
29. Natarajan Shankar. *Proof-checking Metamathematics*. PhD thesis, University of Texas at Austin, 1986.
30. Natarajan Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.
31. Natarajan Shankar. Shankar, Boyer, Church-Rosser and de Bruijn indices. E-mail, 2013.
32. S. Świerczkowski. Finite sets and Gödel's incompleteness theorems. *Dissertationes Mathematicae*, 422:1–58, 2003. `http://journals.impan.gov.pl/dm/Inf/422-0-1.html`.
33. Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.
34. Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. *Logical Methods in Computer Science*, 8(2:14):1–35, 2012.