A Machine-Assisted Proof of Gödel's Incompleteness Theorems

Lawrence C. Paulson, Computer Laboratory, University of Cambridge
The most misunderstood theorems in mathematics

- Gödel’s theorems have highly technical, syntactic proofs.

1. Every “strong enough” formal system is incomplete, in that at least one formula can neither be proved nor disproved.

2. And if such a formal system admits a proof of its own consistency, then it is actually inconsistent.

- For the first time, both of Gödel’s proofs have been mechanised, following a paper by Świerczkowski (2003)

- The machine proof, in the structured Isar language, is complete, almost readable, and can be perused interactively.
Hereditarily finite set theory

- A hereditarily finite set is a finite set of HF sets.
- Many mathematical constructions, including natural numbers and sequences, can be defined as in standard set theory.
- HF set theory is equivalent to Peano Arithmetic via the mapping

\[ f(x) = \sum \{2^{f(y)} \mid y \in x\} \]
Benefits of Using HF Set Theory

• Can use standard definitions of pairing and sequences.

• The first incompleteness theorem requires an HF development of the natural numbers, induction, etc., but not addition.

• The second incompleteness theorem requires operations on sequences and addition, but not multiplication.

• No need for least common multiples, prime numbers or the Chinese remainder theorem.
The Axioms of HF Set Theory

\[ z = 0 \iff \forall x \ [x \notin z] \]
\[ z = x \triangleleft y \iff \forall u \ [u \in z \iff u \in x \lor u = y] \]
\[ \phi(0) \land \forall x y \ [\phi(x) \land \phi(y) \rightarrow \phi(x \triangleleft y)] \rightarrow \forall x \ [\phi(x)] \]

\begin{itemize}
  \item 0 denotes the empty set
  \item \( x \triangleleft y \) denotes the set \( x \) extended with the element \( y \).
  \item There are no other function symbols.
  \item Union, intersection, etc can be shown to exist by induction.
\end{itemize}
Stages of the Proofs

- The *syntax* of a first-order theory is formalised: terms, formulas, substitution...

- A *deductive calculus* for sequents of the form $\Gamma \vdash \alpha$ (typically for Peano arithmetic, but here HF)

- *Meta-theory* to relate truth and provability. E.g. “all true $\Sigma$ formulas are theorems”. $\Sigma$ formulas are built using $\lor \land \exists$ and bounded $\forall$.

- A system of coding to formalise the calculus within itself. The code of $\alpha$ is a term, written $\lceil \alpha \rceil$.

- Syntactic predicates to recognise codes of terms, substitution, axioms, etc.

- Finally the predicate Pf, such that $\vdash \alpha \iff \vdash \text{Pf} \lceil \alpha \rceil$. 
First Incompleteness Theorem

• To prove Gödel’s first incompleteness theorem, construct δ that expresses that δ is not provable.

• It follows (provided the calculus is consistent) that neither δ nor its negation can be proved.

• Need to show that substitution behaves like a function.
  • Requires a detailed proof in the calculus,
  • … alternatively, other detailed calculations.
If $\alpha$ is a $\Sigma$ sentence, then $\vdash \alpha \rightarrow \text{Pf} \lnot \alpha$.

* This crucial lemma for Gödel’s second incompleteness theorem is proved by induction over the construction of $\alpha$ as a $\Sigma$ formula.

* It requires generalising the statement above to allow the formula $\alpha$ to contain free variables.

  * complex technicalities

  * lengthy deductions in the calculus
Proving Theorems in the Calculus

* Gödel knew that formal proofs were difficult. They could be eliminated, but at what cost?

* By coding all predicates as executable functions, and proving a meta-theorem, Gödel reduced provability to truth.

* But then only bounded quantifiers can be used, with tricky arithmetical proofs that the bounds are adequate.

* With Σ formulas, provability is reduced to truth for most desired properties, with no tricky proofs about bounds.

* Instead, some straightforward inductions need to be formalised in the calculus.

* The second theorem requires working in the calculus anyway.
Isabelle/HOL and Nominal

- a proof assistant for higher-order logic
- much automation to hide the underlying proof calculus
- support for recursive functions and inductive sets
- the nominal package, for working with named variables
- Free names are significant, but not once they are bound.
- Syntax involving variable binding can be defined using recursion, provided variables are used “sensibly”.
- During proof by induction, bound variable names can be guaranteed not to clash with specified other terms.
De Bruijn Indexes

* This approach to variable binding replaces names by numbers.
* 0 denotes the innermost bound variable, 1 for the next, etc.
* This approach destroys readability, but substitution and abstraction are very easy to define.
* During coding, formulas are translated into the de Bruijn format.
* And so there is no need to formalise the nominal theory within HF.
Defining Terms and Formulas

Gödel describes a relation \( R(x_1, \ldots, x_n) \) as entscheidungsdefinit (the modern term is numeralwise expressible) provided there is a formula \( R(x_1, \ldots, x_n) \) such that, for each \( x_1, \ldots, x_n \),

\[ R(x_1, \ldots, x_n) \text{ implies } \text{}`
\]

\[ R(x_1, \ldots, x_n) \text{ implies } \neg R(x_1, \ldots, x_n) \text{ (2) }
\]

Here, \( \text{}`\) means "not \( R\)" and \( x_1, \ldots, x_n \) denotes the numerals expressing the values of \( x_1, \ldots, x_n \) (? , p. 130). This technique shows that \( R(x_1, \ldots, x_n) \) is a theorem of the formal calculus without requiring an explicit proof.

1 The Isabelle/HOL proof development: fundamentals.

```plaintext
nominal_datatype \( \text{tm} = \) Zero | Var name | Eats \( \text{tm} \) \( \text{tm} \)

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

Variable binding formalised using nominal
Defining Substitution

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)"

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))"

The variable j must be fresh for i and x

Properties of substitution have simple proofs.
Defining the HF Calculus

For substitution within a formula, we normally expect issues concerning the capture of a bound variable. Note that the result of substituting the term \( x \) for the variable \( i \) in the formula \( A \) is written \( A(i::=x) \).

```
primrec subst :: "fm + name \to tm \to 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 \(\notin\) (i, x) \(\equiv\) (Ex j A)(i::=x) = Ex j (A(i::=x))"
```

Substitution is again straightforward in the first four cases (membership, equality, disjunction, negation). In the existential case, the precondition \( atom j \not\in (i, x) \) (pronounced "\( j \) is fresh for \( i \) and \( x \)"") essentially says that \( i \) and \( j \) must be different names with \( j \) not free in \( x \). We do not need to supply a mechanism for renaming the bound variable, as that is part of the nominal framework, which in most cases will choose a sufficiently fresh bound variable at the outset. The usual properties of substitution (commutativity, for example) have simple proofs by induction on formulas. In contrast, needed to combine three substitution lemmas in a simultaneous proof by induction, a delicate argument involving 1900 lines of Coq.

The HF proof system is an inductively defined predicate, where \( H \vdash A \) means that the formula \( A \) is provable from the set of formulas \( H \).

```
inductive hfthm :: "fm set \to fm \to bool" (infixl "\vdash" 55) where
  Hyp: "A \in H \To H \vdash A"
| Extra: "H \vdash extra_axiom"
| Bool: "A \in boolean_axioms \To H \vdash A"
| Eq: "A \in equality_axioms \To H \vdash A"
| Spec: "A \in special_axioms \To H \vdash A"
| HF: "A \in HF_axioms \To H \vdash A"
| Ind: "A \in induction_axioms \To H \vdash A"
| MP: "H \vdash A IMP B \To H' \vdash A \To H \cup H' \vdash B"
| Exists: "H \vdash A IMP B \To
  atom i \not\in B \To \forall C \in H. atom i \not\in C \To H \vdash (Ex i A) IMP B"
```

The variable \( i \) must be fresh for \( B \) and \( H \).
Early Steps in the HF Calculus

* the deduction theorem (yielding a sequent calculus)

* derived rules to support explicit formal proofs
  
  * for defined connectives, including \( \land \rightarrow \forall \)

  * for equality, set induction, …

* definitions and proofs for subsets, extensionality, foundation and natural number induction
Σ Formulas

Strict Σ formulas only contain variables and are the basis for the main induction of the second incompleteness theorem. We can still derive the general case of Σ formulas.

```isabelle
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)"

"Sigma_fm A ⇐ (∃B. ss_fm B & supp B ⊆ supp A & {}) ⊢ A IFF B)"

theorem "[Sigma_fm A; ground_fm A; eval_fm e0 A] ⊢ {A}"

True Σ formulas are theorems!
```
Coding Terms and Formulas

- must first translate from nominal to de Bruijn format
  - the actual coding is a simple recursive map:
    - ⌜0⌟ = 0, ⌜x⌝ = k, ⌜x ⊲ y⌟ = ⟨⌜x⌟, ⌜x⌟, ⌜y⌟⟩, ...
  - also define (in HF) predicates to recognise codes
    - abstraction over a variable — needed to define Form(x), the predicate for formulas
    - substitution for a variable
Example: Making a Formula

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’)

    y = u ∨ w, or y = ¬ u, or y = (∃v) u
    with an explicit abstraction step on 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)))"

The “official” version as an HF formula, not a boolean
Those Coding Predicates

- SeqTerm
- Term
- SeqConst
- Const
- SeqStTerm
- AbstTerm
- SubstTerm
- AbstAtomic
- SeqAbstForm
- AbstForm
- SubstAtomic
- SeqSubstForm
- SubstForm
- Atomic
- MakeForm
- SeqForm
- Form
- VarNonOccTerm
- VarNonOccAtomic
- SeqVarNonOccForm
- VarNonOccForm
... And Proof Predicates

(a sequence of proof steps, and finally...)
Steps to the First Theorem

- We need a function $K$ such that $\vdash K(\neg \phi) = \neg \phi(\neg \phi)$

- … but we have no function symbols. Instead, define a relation, $KRP$:
  
  lemma prove_KRP: "{} $\vdash KRP [\forall i ~ [A \land A(i::=\neg A)]]$

- Proving its functional behaviour takes 600 HF proof steps.
  
  lemma KRP_unique: "\{KRP \lor x \land y, KRP \lor x \land y'\} $\vdash y' \land EQ y$

- Finally, the diagonal lemma:
  
  lemma diagonal:
  
  obtains $\delta$ where "{} $\vdash \delta \iff \alpha(i::=\neg \delta)" ~ "supp \delta = supp \alpha - \{atom \ i\}"
theorem Goedel_I:
  assumes Con: "¬ { } ⊨ Fls"
  obtains δ where "{ } ⊨ δ IFF Neg (PfP «δ»)"
      "¬ { } ⊨ δ" "¬ { } ⊨ Neg δ"
      "eval_fm e δ" "ground_fm δ"

proof -
  obtain δ where "{ } ⊨ δ IFF Neg ((PfP (Var i))(i:=«δ»))"
      and [simp]: "supp δ = supp (Neg (PfP (Var i))) - {atom i}"
      by (metis SyntaxN.Neg diagonal)
  hence diag: "{ } ⊨ δ IFF Neg (PfP «δ»)"
      by simp
  hence np: "¬ { } ⊨ δ"
      by (metis Con Iff_MP_same Neg_D proved_iff_proved_Pf)
  hence npn: "¬ { } ⊨ Neg δ" using diag
      by (metis Iff_MP_same NegNeg_D Neg_cong proved_iff_proved_Pf)
  moreover have "eval_fm e δ" using hfthm_sound [where e=e, OF diag]
      by simp (metis Pf_quot_imp_is_proved np)
  moreover have "ground_fm δ"
      by (auto simp: ground_fm_aux_def)
  ultimately show ?thesis
      by (metis diag np npn that)
qed
Steps to the Second Theorem

• Coding must be generalised to allow variables in codes.

  • \( \langle r \diamond \gamma, r x, r y \rangle \)

  • \( \langle r \diamond \gamma, x, y \rangle \)

• Variables must be renamed, with the intent of creating “pseudo-terms” like \( \langle r \diamond \gamma, Q x, Q y \rangle \).

• Q is a magic function: \( Q x = r t \gamma \) where \( t \) is some canonical term denoting the set \( x \).
Complications

- Q must be a relation.
  - Function symbols cannot be added…
  - Sets do not have an easily defined canonical ordering.

- QR(0,0)

- QR(x,x’), QR(y,y’) \implies QR(x \triangleleft y, \langle r \triangleleft \neg, x’, y’ \rangle)
One of the Final Lemmas

\[
\begin{align*}
\text{QR}(x, x'), \text{QR}(y, y') & \vdash x \in y \rightarrow \text{Pf} [x' \in y']_{\{x', y'\}} \\
\text{QR}(x, x'), \text{QR}(y, y') & \vdash x \subseteq y \rightarrow \text{Pf} [x' \subseteq y']_{\{x', y'\}} \\
\text{QR}(x, x'), \text{QR}(y, y') & \vdash x = y \rightarrow \text{Pf} [x' = y']_{\{x', y'\}}
\end{align*}
\]

- The first two require simultaneous induction, yielding the third.
- Similar proofs for the symbols \(\lor, \land, \exists\) and bounded \(\forall\).
- The proof in the HF calculus needs under 450 lines.
- Fills a major gap in various presentations, including Świerczkowski's.
theorem Goedel_II:
  assumes Con: "¬ \{\} \vdash Fls"
  shows "¬ \{\} \vdash Neg (PfP \{Fls\})"
proof -
  from Con Goedel_I obtain \(\delta\)
  where diag: "\{\} \vdash \delta \text{ IFF } Neg (PfP \{\delta\})" "¬ \{\} \vdash \delta"
          and gnd: "ground_fm \(\delta\)"
  by metis
  have "{PfP \{\delta\}} \vdash PfP \{PfP \{\delta\}\}"
  by (auto simp: Provability ground_fm_aux_def supp_conv_fresh)
  moreover have "{PfP \{\delta\}} \vdash PfP \{Neg (PfP \{\delta\})\}"
  apply (rule MonPon_PfP_implies_PfP [OF _ gnd])
  apply (auto simp: ground_fm_aux_def supp_conv_fresh) using diag
  by (metis Assume ContraProve Iff_MP_left Iff_MP_left' Neg_Neg_iff)
  moreover have "ground_fm (PfP \{\delta\})"
  by (auto simp: ground_fm_aux_def supp_conv_fresh)
  ultimately have "{PfP \{\delta\}} \vdash PfP \{Fls\}"
  using PfP_quot_contra
  by (metis (no_types) anti_deduction cut2)
  thus "¬ \{\} \vdash Neg (PfP \{Fls\})"
  by (metis Iff_MP2_same Neg_mono_mono cut1 diag)
qed
What Did We Learn?

- Some highly compressed proofs were finally made explicit.
- The entire proof development can be examined interactively.
- The nominal package can cope with very large developments…
  (BUT: performance issues, some repetitive notation, complications in accepting function definitions)
- <9 months for the first theorem, a further 4 for the second
- Under 16 000 lines of proof script in all.
Conclusions

- the first-ever machine formalisation of Gödel’s second incompleteness theorem
- using both nominal and de Bruijn syntax for bound variables
- within an axiom system for hereditarily finite set theory
- conducted using Isabelle/HOL.

Many thanks to Christian Urban for help at critical points!

Also Jesse Alama, Dana Scott.