Essential Incompleteness of Arithmetic Verified by Coq

Russell O'Connor

[1] Institute for Computing and Information Science,
Faculty of Science, Radboud University Nijmegen
[2] The Group in Logic and the Methodology of Science,
University of California, Berkeley
r.oconnor@cs.ru.nl

Abstract. A constructive proof of the Gödel-Rosser incompleteness theorem [9] has been completed using the Coq proof assistant. Some theory of classical first-order logic over an arbitrary language is formalized. A development of primitive recursive functions is given, and all primitive recursive functions are proved to be representable in a weak axiom system. Formulas and proofs are encoded as natural numbers, and functions operating on these codes are proved to be primitive recursive. The weak axiom system is proved to be essentially incomplete. In particular, Peano arithmetic is proved to be consistent in Coq's type theory and therefore is incomplete.

1 Introduction

The Gödel-Rosser incompleteness theorem for arithmetic states that any complete first-order theory of a nice axiom system, using only the symbols $+$, $\times$, $\mathbf{0}$, $\boldsymbol{S}$, and $<$ is inconsistent. A nice axiom system must contain the nine specific axioms of a system called NN. These nine axioms serve to define the previous symbols. A nice axiom system must also be expressible in itself. This last restriction prevents the incompleteness theorem from applying to axioms systems such as the true first order statements about $\mathbb{N}$.

A computer verified proof of Gödel's incompleteness theorem is not new. In 1986 Shankar created a proof of the incompleteness of Z2, hereditarily finite set theory, in the Boyer-Moore theorem prover [11]. My work is the first computer verified proof of the essential incompleteness of arithmetic. Harrison recently completed a proof in HOL Light [6] of the essential incompleteness of Σ_1-complete theories, but has not shown that any particular theory is Σ_1-complete. His work will be included in the next release of HOL Light.

My proof was developed and checked in Coq 7.3.1 using Proof General under XEmacs. It is part of the user contributions to Coq and can be checked in Coq 8.0 [14]. Examples of source code in this document use the new Coq 8.0 notation.

Coq is an implementation of the calculus of (co)inductive constructions. This dependent type theory has intensional equality and is constructive, so my proof is

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 245–260, 2005.

constructive. Actually the proof depends on the Ensembles library which declares an axiom of extensionality for `Ensembles`, but this axiom is never used.

This document points out some of the more interesting problems I encountered when formalizing the incompleteness theorem. My proof mostly follows the presentation of incompleteness given in *An Introduction to Mathematical Logic* [10]. I referred to the supplementary text for the book *Logic for Mathematics and Computer Science* [1] to construct Gödel's β-function. I also use part of Caprotti and Oostdijk's contribution of Pocklington's criterion [2] to prove the Chinese remainder theorem.

This document is organized as follows. First I discuss the difficulties I had when formalizing classical first-order logic over an arbitrary language. This is followed by the definition of a language LNN and an axiom system called NN. Next I give the statement of the essential incompleteness of NN. Then I briefly discuss coding formulas and proofs as natural numbers. Next I discuss primitive recursive functions and the problems I encountered when trying to prove that substitution can be computed by a primitive recursive function. Finally I briefly discuss the fixed-point theorem, Rosser's incompleteness theorem, and the incompleteness of PA. At the end I give some remarks about how to extend my work in order to formalize Gödel's second incompleteness theorem.

1.1 Coq Notation

For those not familiar with Coq syntax, here is a short list of notation

- `->`, `/\`, `\/`, and `~` are the logical connectives $\Rightarrow$, $\wedge$, $\vee$, and $\neg$.
- `A -> B`, `A * B`, and `A + B` form function types, Cartesian product types, and disjoint union types.
- `*`, `+`, and `S` are the arithmetic operations of multiplication, addition, and successor.
- `inl` and `inr` are the left and right injection functions of types `A -> A + B` and `B -> A + B`.
- `::`, and `++` are the list operations cons, and append.
- `_` is an omitted parameter that Coq can infer itself.

For more details see the Coq 8.0 reference manual [14].

2 First-Order Classical Logic

I began by developing the theory of first order classical logic inside Coq. In essence Coq's logic is a formal metalogic to reason about this internal logic.

2.1 Definition of `Language`

I immediately took advantage of Coq's dependent type system by defining `Language` to be a dependent record of types for symbols and an arity function from symbols to $\mathbb{N}$. The Coq code is:

```
Record Language : Type := language
  {Relations : Set;
   Functions : Set;
   arity : Relations + Functions -> nat}.
```

In retrospect it would have been slightly more convenient to use two arity functions instead of using the disjoint union type.

This approach differs from Harrison's definition of first order terms and formulas in HOL Light [5] because HOL Light does not have dependent types. Dependent types allow the type system to enforce that all terms and formulas of a given language are well formed.

2.2 Definition of `Term`

For any given language, a `Term` is either a variable indexed by a natural number or a function symbol plus a list of n terms where n is the arity of the function symbol. My first attempt at writing this in Coq failed.

```
Variable L : Language.
(* Invalid definition *)
Inductive Term0 : Set :=
  | var0 : nat -> Term0
  | apply0 : forall (f : Functions L) (l : List Term0),
       (arity L (inr _ f))=(length l) -> Term0.
```

The type `(arity L (inr _ f))=(length l)` fails to meet Coq's positivity requirement for inductive types. Expanding the definition of `length` reveals a hidden occurrence of `Term0` which is passed as an implicit argument to `length`. It is this occurrence that violates the positivity requirement.

My second attempt met the positivity requirement, but it had other difficulties. A common way to create a polymorphic lists of length `n` is:

```
Inductive Vector (A : Set) : nat -> Set :=
  | Vnil : Vector A 0
  | Vcons : forall (a : A) (n : nat),
       Vector A n -> Vector A (S n).
```

Using this I could have defined `Term` like:

```
Variable L : Language.

Inductive Term1 : Set :=
  | var1 : nat -> Term1
  | apply1 : forall f : Functions L,
       (Vector Term1 (arity L (inr _ f))) -> Term1.
```

My difficulty with this definition was that the induction principle generated by Coq is too weak to work with.

Instead I created two mutually inductive types: `Term` and `Terms`.

```
Variable L : Language.

Inductive Term : Set :=
  | var : nat -> Term
  | apply : forall f : Functions L,
       Terms (arity L (inr _ f)) -> Term
with Terms : nat -> Set :=
  | Tnil : Terms 0
  | Tcons : forall n : nat,
       Term -> Terms n -> Terms (S n).
```

Again the automatically generated induction principle is too weak, so I used the `Scheme` command to generate suitable mutual-inductive principles.

The disadvantage of this approach is that useful lemmas about `Vector`s must be reproved for `Term`s. Some of these lemmas are quite tricky to prove because of the dependent type. For example, proving `forall x : Terms 0, Tnil = x` is not easy.

Recently, Marche has shown me that the `Term1` definition would be adequate. One can explicitly make a sufficient induction principle by using nested `Fixpoint` functions [7].

2.3 Definition of `Formula`

The definition of `Formula` was straightforward.

```
Inductive Formula : Set :=
  | equal : Term -> Term -> Formula
  | atomic : forall r : Relations L, Terms (arity L (inl _ r)) ->
                Formula
  | impH : Formula -> Formula -> Formula
  | notH : Formula -> Formula
  | forallH : nat -> Formula -> Formula.
```

I defined the other logical connectives in terms of `impH`, `notH`, and `forallH`.

The `H` at the end of the logic connectives (such as `impH`) stands for "Hilbert" and is used to distinguish them from Coq's connectives.

For example, the formula $\neg\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.\boldsymbol{x}_0 = \boldsymbol{x}_1$ would be represented by:

```
notH (forallH 0 (forallH 1 (equal (var 0) (var 1))))
```

It would be nice to use higher order abstract syntax to handle bound variables by giving `forallH` the type `(Term -> Formula) -> Formula`. I would represent the above example as:

```
notH (forallH (fun x : Term =>
        (forallH (fun y : Term => (equal x y)))))
```

This technique would require addition work to disallow "exotic terms" that are created by passing a function into `forallH` that does a case analysis on the term

and returning entirely different formulas in different cases. Despeyroux et al. [3] address this problem by creating a complicated predicate that only valid formulas satisfy.

Another choice would have been to use de Bruijn indexes to eliminate named variables. However dealing with free and bound variables with de Bruijn indexes can be difficult.

Using named variables allowed me to closely follow Hodel's work [10]. Also, in order to help persuade people that the statement of the incompleteness theorem is correct, it is helpful to make the underlying definitions as familiar as possible.

Renaming bound variables turned out to be a constant source of work during development because variable names and terms were almost always abstract. In principle the variable names could conflict, so it was constantly necessary to consider this case and deal with it by renaming a bound variable to a fresh one. Perhaps it would have been better to use de Bruijn indexes and a deduction system that only deduced closed formulas.

2.4 Definition of `substituteFormula`

I defined the function `substituteFormula` to substitute a term for all occurrences of a free variable inside a given formula. While the definition of `substituteTerm` is simple structural recursion, substitution for formulas is complicated by quantifiers. Suppose we want to substitute the term s for $\boldsymbol{x}_i$ in the formula $\forall \boldsymbol{x}_j.\varphi$ and $i \neq j$. Suppose $\boldsymbol{x}_j$ is a free variable of s. If we naïvely perform the substitution then the occurrences of $\boldsymbol{x}_j$ in s get captured by the quantifier. One common solution to this problem is to disallow substitution for a term s when s is not *substitutable* for x_i in φ. The solution I take is to rename the bound variable in this case.

$$(\forall \boldsymbol{x}_j.\varphi)[\boldsymbol{x}_i/s] \stackrel{\text{def}}{=} \forall \boldsymbol{x}_k.(\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k])[\boldsymbol{x}_i/s] \text{ where } k \neq i \text{ and } \boldsymbol{x}_k \text{ is not free in } \varphi \text{ or } s$$

Unfortunately this definition is not structurally recursive. The second substitution operates on the result of the first substitution, which is not structurally smaller than the original formula.

Coq will not accept this recursive definition as is; it is necessary to prove the recursion will terminate. I proved that substitution preserves the *depth* of a formula, and that each recursive call operates on a formula of smaller depth.

One of McBride's mantras says, "If my recursion is not structural, I am using the wrong structure" [8, p. 241]. In this case, my recursion is not structural because I am using the wrong recursion. Stoughton shows that it is easier to define substitution that substitutes all variables simultaneously because the recursion is structural [13]. If I had made this definition first, I could have defined substitution of one variable in terms of it and many of my difficulties would have disappeared.

2.5 Definition of `Prf`

I defined the inductive type `(Prf Gamma phi)` to be the type of proofs of `phi`, from the list of assumptions `Gamma`.

```
Inductive Prf : Formulas -> Formula -> Set :=
  | AXM : forall A : Formula, Prf (A :: nil) A
  | MP : forall (Axm1 Axm2 : Formulas) (A B : Formula),
      Prf Axm1 (impH A B) -> Prf Axm2 A ->
         Prf (Axm1 ++ Axm2) B
  | GEN : forall (Axm : Formulas) (A : Formula) (v : nat),
      ~ In v (freeVarListFormula L Axm) -> Prf Axm A ->
         Prf Axm (forallH v A)
  | IMP1 : forall A B : Formula, Prf nil (impH A (impH B A))
  | IMP2 : forall A B C : Formula,
      Prf nil (impH (impH A (impH B C))
                    (impH (impH A B) (impH A C)))
  | CP : forall A B : Formula,
      Prf nil (impH (impH (notH A) (notH B)) (impH B A))
  | FA1 : forall (A : Formula) (v : nat) (t : Term),
      Prf nil (impH (forallH v A) (substituteFormula L A v t))
  | FA2 : forall (A : Formula) (v : nat),
      ~ In v (freeVarFormula L A) ->
         Prf nil (impH A (forallH v A))
  | FA3 : forall (A B : Formula) (v : nat),
      Prf nil
        (impH (forallH v (impH A B))
           (impH (forallH v A) (forallH v B)))
  | EQ1 : Prf nil (equal (var 0) (var 0))
  | EQ2 : Prf nil (impH (equal (var 0) (var 1))
                        (equal (var 1) (var 0)))
  | EQ3 : Prf nil
        (impH (equal (var 0) (var 1))
           (impH (equal (var 1) (var 2)) (equal (var 0) (var 2))))
  | EQ4 : forall R : Relations L, Prf nil (AxmEq4 R)
  | EQ5 : forall f : Functions L, Prf nil (AxmEq5 f).
```

`AxmEq4` and `AxmEq5` are recursive functions that generate the equality axioms for relations and functions. `AxmEq4 R` generates

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 \Rightarrow \ldots \Rightarrow \boldsymbol{x}_{2n-2} = \boldsymbol{x}_{2n-1} \Rightarrow (R(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{2n-2}) \Leftrightarrow R(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{2n-1}))$$

and `AxmEq5 f` generates

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 \Rightarrow \ldots \Rightarrow \boldsymbol{x}_{2n-2} = \boldsymbol{x}_{2n-1} \Rightarrow f(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{2n-2}) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{2n-1})$$

I found that replacing ellipses from informal proofs with recursive functions was one of the most difficult tasks. The informal proof does not contain information on what inductive hypothesis should be used when reasoning about these recursive definitions. Figuring out the correct inductive hypotheses was not easy.

2.6 Definition of `SysPrf`

There are some problems with the definition of `Prf` given. It requires the list of axioms to be in the correct order for the proof. For example, if we have `Prf Gamma1 (impH phi psi)` and `Prf Gamma2 phi` then we can conclude only `Prf Gamma1++Gamma2 psi`. We cannot conclude `Prf Gamma2++Gamma1 psi` or any other permutation of `psi`. If an axiom is used more than once, it must appear in the list more than once. If an axiom is never used, it must not appear. Also, the number of axioms must be finite because they form a list.

To solve this problem, I defined a `System` to be `Ensemble Formula`, and `(SysPrf T phi)` to be the proposition that the system `T` proves `phi`.

```
Definition System := Ensemble Formula.
Definition mem := Ensembles.In.

Definition SysPrf (T : System) (f : Formula) :=
  exists Axm : Formulas,
    (exists prf : Prf Axm f,
       (forall g : Formula, In g Axm -> mem _ T g)).
```

`Ensemble A` represents subsets of `A` by the functions `A -> Prop`. `a : A` is considered to be a member of `T : Ensemble A` if and only if the type `T a` is inhabited. I also defined `mem` to be `Ensembles.In` so that it does not conflict with `List.In`.

2.7 The Deduction Theorem

The deduction theorem states that if $\Gamma \cup \{\varphi\} \vdash \psi$ then $\Gamma \vdash \varphi \Rightarrow \psi$.

There is a choice of whether the side condition for the $\forall$-generalization rule, `~ In v (freeVarListFormula L Axm)`, should be required or not. If this side condition is removed then the deduction theorem requires a side condition on it. Usually all the formulas in an axiom system are closed, so the side condition on the $\forall$-generalization is easy to show. So I decided to keep the side condition on the $\forall$-generalization rule.

At one point the proof of the deduction theorem requires proving that if $\Gamma \cup \{\varphi\} \vdash \psi$ because $\psi \in \Gamma \cup \{\varphi\}$, then $\Gamma \vdash \varphi \Rightarrow \psi$. There are two cases to consider. If $\psi = \varphi$ then the result easily follows from the reflexivity of $\Rightarrow$. Otherwise $\psi \in \Gamma$, and therefore $\Gamma \vdash \psi$. The result then follows. In order to constructively make this choice it is necessary to decide whether $\psi = \varphi$ or not. This requires `Formula` to be a decidable type, and that requires the language `L` to be decidable. Since `L` could be anything, I needed to add hypotheses that the function and relation symbols are decidable types.

```
- forall x y : Functions L, { x=y } + { x<>y }
- forall x y : Relations L, { x=y } + { x<>y }.
```

I used the deduction theorem without restriction and ended up using the hypotheses in many lemmas. I expect that many of these lemmas could be proved without assuming the decidability of the language. It is hard to imagine a useful language that is not decidable, so I do not feel too bad about using these hypotheses in unnecessary places.

2.8 Languages and Theories of Number Theory

I created two languages. The first language, `LNT`, is the language of number theory and just has the function symbols `Plus`, `Times`, `Succ`, and `Zero` with appropriate arities. The second language, `LNN`, is the language of NN and has the same function symbols as `LNT` plus one relation symbol for less than, `LT`.

I define two axiom systems: `NN` and `PA`. `NN` and `PA` share six axioms.

1. $\forall \boldsymbol{x}_0.\neg \boldsymbol{S}\boldsymbol{x}_0 = \boldsymbol{0}$
2. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.(\boldsymbol{S}\boldsymbol{x}_0 = \boldsymbol{S}\boldsymbol{x}_1 \Rightarrow \boldsymbol{x}_0 = \boldsymbol{x}_1)$
3. $\forall \boldsymbol{x}_0.\boldsymbol{x}_0 + \boldsymbol{0} = \boldsymbol{x}_0$
4. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.\boldsymbol{x}_0 + \boldsymbol{S}\boldsymbol{x}_1 = \boldsymbol{S}(\boldsymbol{x}_0 + \boldsymbol{x}_1)$
5. $\forall \boldsymbol{x}_0.\boldsymbol{x}_0 \times \boldsymbol{0} = \boldsymbol{0}$
6. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.\boldsymbol{x}_0 \times \boldsymbol{S}\boldsymbol{x}_1 = (\boldsymbol{x}_0 \times \boldsymbol{x}_1) + \boldsymbol{x}_0$

`NN` has three additional axioms about less than.

1. $\forall \boldsymbol{x}_0.\neg \boldsymbol{x}_0 < \boldsymbol{0}$
2. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.(\boldsymbol{x}_0 < \boldsymbol{S}\boldsymbol{x}_1 \Rightarrow (\boldsymbol{x}_0 = \boldsymbol{x}_1 \vee \boldsymbol{x}_0 < \boldsymbol{x}_1))$
3. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.(\boldsymbol{x}_0 < \boldsymbol{x}_1 \vee \boldsymbol{x}_0 = \boldsymbol{x}_1 \vee \boldsymbol{x}_1 < \boldsymbol{x}_0)$

`PA` has an infinite number of induction axioms that follow one schema.

1. (schema) $\forall \boldsymbol{x}_{i_1}. \ldots \forall \boldsymbol{x}_{i_n}.\varphi[\boldsymbol{x}_j/\boldsymbol{0}] \Rightarrow \forall \boldsymbol{x}_j.(\varphi \Rightarrow \varphi[\boldsymbol{x}_j/\boldsymbol{S}\boldsymbol{x}_j]) \Rightarrow \forall \boldsymbol{x}_j.\varphi$

The $\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_n}$ are the free variables of $\forall \boldsymbol{x}_j.\varphi$. The quantifiers ensure that all the axioms of PA are closed.

Because NN is in a different language than PA, a proof in NN is not a proof in PA. In order to reuse the work done in NN, I created a function called `LNN2LNT_formula` to convert formulas in `LNN` into formulas in `LNT` by replacing occurrences of $t_0 < t_1$ with $(\exists \boldsymbol{x}_2.\boldsymbol{x}_0 + (\boldsymbol{S}\boldsymbol{x}_2) = \boldsymbol{x}_1)[\boldsymbol{x}_0/t_0, \boldsymbol{x}_1/t_1]$—$\varphi[\boldsymbol{x}_0/t_0, \boldsymbol{x}_1/t_1]$ is the simultaneous substitution of t_0 for $\boldsymbol{x}_0$ and t_1 for $\boldsymbol{x}_1$. Then I proved that if $\mathrm{NN} \vdash \varphi$ then $\mathrm{PA} \vdash \texttt{LNN2LNT\_formula}(\varphi)$.

I also created the function `natToTerm : nat -> Term` to return the closed term representing a given natural number. In this document I will refer to this function as $\ulcorner \boldsymbol{.} \urcorner$, so $\ulcorner 0 \urcorner = \boldsymbol{0}$, $\ulcorner 1 \urcorner = \boldsymbol{S0}$, etc.

3 Coding

To prove the incompleteness theorem, it is necessary for the inner logic to reason about proofs and formulas, but the inner logic can only reason about natural numbers. It is therefore necessary to code proofs and formulas as natural numbers.

Gödel's original approach was to code a formula as a list of numbers and then code that list using properties from the prime decomposition theorem[4]. I avoided needing theorems about prime decomposition by using the Cantor pairing function instead. The Cantor pairing function, `cPair`, is a commonly used bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

$$\texttt{cPair}(a, b) \stackrel{\text{def}}{=} a + \sum_{i=1}^{a+b} i$$

All my inductive structures were easy to recursively encode. I gave each constructor a unique number and paired that number with the encoding of all its parameters. For example, I defined `codeFormula` as:

```
Fixpoint codeFormula (f : Formula) : nat :=
  match f with
  | fol.equal t1 t2 => cPair 0 (cPair (codeTerm t1) (codeTerm t2))
  | fol.impH f1 f2 =>
       cPair 1 (cPair (codeFormula f1) (codeFormula f2))
  | fol.notH f1 => cPair 2 (codeFormula f1)
  | fol.forallH n f1 => cPair 3 (cPair n (codeFormula f1))
  | fol.atomic R ts => cPair (4+(codeR R)) (codeTerms _ ts)
  end.
```

where `codeR` is a coding of the relation symbols for the language.

I will use $\ulcorner\varphi\urcorner$ for $\ulcorner$`codeFormula` $\varphi\urcorner$ and $\ulcorner t\urcorner$ for $\ulcorner$`codeTerm` $t\urcorner$.

4 The Statement of Incompleteness

The incompleteness theorem states the essential incompleteness of NN, meaning that for every axiom system T such that

- NN $\subseteq T$
- T can represent its own axioms
- T is a decidable set

then there exists a sentence φ such that if $T \vdash \varphi$ or $T \vdash \neg\varphi$ then T is inconsistent.

The theorem is only about proofs in `LNN`, the language of NN. This statement does not show the incompleteness of theories that extend the language.

In Coq the theorem is stated as as:

```
Theorem Incompleteness
     : forall T : System,
       Included Formula NN T ->
       RepresentsInSelf T ->
       DecidableSet Formula T ->
       exists f : Formula,
         Sentence f /\
         (SysPrf T f \/ SysPrf T (notH f) -> Inconsistent LNN T).
```

A `System` is `Inconsistent` if it proves all formulas.

```
Definition Inconsistent (T : System) :=
   forall f : Formula, SysPrf T f.
```

A `Sentence` is a `Formula` without any free variables.

```
Definition Sentence (f : Formula) :=
   forall v : nat, ~ In v (freeVarFormula LNN f).
```

A `DecidableSet` is an `Ensemble` such that every item either belongs to the `Ensemble` or does not belong to the `Ensemble`. This hypothesis is trivially true in classical logic, but in constructive logic I needed it to prove the strong constructive existential quantifier in the statement of incompleteness.

```
Definition DecidableSet (A : Type)(s : Ensemble A) :=
   forall x : A, mem A s x \/ ~ mem A s x.
```

The `RepresentsInSelf` hypothesis restricts what the `System` T can be. The statement of essential incompleteness normally requires T be a recursive set. Instead I use the weaker hypothesis that the set T is expressible in the system T.

Given a system T extending NN and another system U along with a formula φ_U with at most one free variable $\boldsymbol{x}_i$, we say φ_U expresses the axiom system U in T if the following hold for all formulas ψ.

1. if $\psi \in U$ then $T \vdash \varphi_U[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$
2. if $\psi \notin U$ then $T \vdash \neg\varphi_U[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$

U is expressible in T if there exists a formula φ_U such that φ_U expresses the axiom system U in T.

In Coq I write the statement T is expressible in T as

```
Definition RepresentsInSelf (T : System) :=
exists rep : Formula, exists v : nat,
(forall x : nat, In x (freeVarFormula LNN rep) -> x = v)  /\
(forall f : Formula,
        mem Formula T f ->
        SysPrf T (substituteFormula LNN rep v
                    (natToTerm (codeFormula f)))) /\
(forall f : Formula,
        ~ mem Formula T f ->
        SysPrf T (notH (substituteFormula LNN rep v
                          (natToTerm (codeFormula f))))).
```

This is weaker than requiring that T be a recursive set because any recursive set of axioms T is expressible in NN. Since T is an extension of NN, any recursive set of axioms T is expressible in T.

By using this weaker hypothesis I avoid defining what a recursive set is. Also, in this form the theorem could be used to prove that any complete and consistent theory of arithmetic cannot define its own axioms. In particular, this could be used to prove Tarski's theorem that the truth predicate is not definable.

5 Primitive Recursive Functions

A common approach to proving the incompleteness theorem is to prove that every primitive recursive function is representable. Informally an n-ary function f is representable in NN if there exists a formula φ such that

1. the free variables of φ are among $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_n$.
2. for all $a_1, \ldots, a_n : \mathbb{N}$,
   $\mathrm{NN} \vdash (\varphi \Rightarrow \boldsymbol{x}_0 = \ulcorner f(a_1, \ldots, a_n) \urcorner)[\boldsymbol{x}_1/\ulcorner a_1 \urcorner, \ldots, \boldsymbol{x}_n/\ulcorner a_n \urcorner]$

I defined the type `PrimRec n` as:

```
Inductive PrimRec : nat -> Set :=
  | succFunc : PrimRec 1
  | zeroFunc : PrimRec 0
  | projFunc : forall n m : nat, m < n -> PrimRec n
  | composeFunc :
      forall (n m : nat) (g : PrimRecs n m) (h : PrimRec m),
        PrimRec n
  | primRecFunc :
      forall (n : nat) (g : PrimRec n) (h : PrimRec (S (S n))),
        PrimRec (S n)
with PrimRecs : nat -> nat -> Set :=
  | PRnil : forall n : nat, PrimRecs n 0
  | PRcons : forall n m : nat,
      PrimRec n -> PrimRecs n m -> PrimRecs n (S m).
```

`PrimRec n` is the expression of an n-ary primitive recursive function, but it is not itself a function. I defined `evalPrimRec : forall n : nat, PrimRec n -> naryFunc n` to convert the expression into a function. Rather than working directly with primitive recursive expressions, I worked with particular Coq functions and proved they were extensionally equivalent to the evaluation of primitive recursive expressions.

I proved that every primitive recursive function is representable in NN. This required using Gödel's β-function along with the Chinese remainder theorem. The β-function is a function that codes array indexing. A finite list of numbers $a_0, \ldots, a_n$ is coded as a pair of numbers (x, y) and $\beta(x, y, i) = a_i$. The β-function is special because it is defined in terms of plus and times and is non-recursive. The Chinese remainder theorem is used to prove that the β-function works.

I took care to make the formulas representing the primitive recursive functions clearly Σ_1 by ensuring that only the unbounded quantifiers are existential; however, I did not prove that the formulas are Σ_1 because it is not needed for the first incompleteness theorem. Such a proof could be used for the second incompleteness theorem [12].

5.1 `codeSubFormula` Is Primitive Recursive

I proved that substitution is primitive recursive. Since substitution is defined in terms of `Formula` and `Term`, it itself cannot be primitive recursive. Instead I proved that the corresponding function operating on codes is primitive recursive. This function is called `codeSubFormula` and I proved it is correct in the following sense.

$$\texttt{codeSubFormula}(\ulcorner \varphi \urcorner, i, \ulcorner s \urcorner) = \ulcorner \varphi[\boldsymbol{x}_i/s] \urcorner$$

Next I proved that it is primitive recursive. This proof is very difficult. The problem is again with the need to rebind bound variables. Normally one would attempt to create this primitive recursive function by using course-of-values recursion. Course-of-values recursion requires all recursive calls have a smaller code than the original call. Renaming a bound variable requires two recursive calls. Recall the definition of substitution in this case:

$$(\forall \boldsymbol{x}_j.\varphi)[\boldsymbol{x}_i/s] \stackrel{\text{def}}{=} \forall \boldsymbol{x}_k.(\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k])[\boldsymbol{x}_i/s] \text{ where } k \neq i \text{ and } \boldsymbol{x}_k \text{ is not free in } \varphi \text{ or } s$$

If one is lucky one might be able to make the inner recursive call. But there is no reason to suspect the input to the second recursive call, $\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k]$, is going to have a smaller code than the original input, $\forall \boldsymbol{x}_j.\varphi$.

If I had used the alternative definition of substitution, where all variables are substituted simultaneously, there would still be problems. The input would include a list of variable and term pairs. In this case a new pair would be added to the list when making the recursive call, so the input to the recursive call could still have a larger code than the input to the original call.

It seems that using course-of-values recursion is difficult or impossible. Instead I introduce the notion of the trace of the computation of substitution. Think of the trace of computation as a finite tree where the nodes contain the input and output of each recursive call. The subtrees of a node are the traces of the computation of the recursive calls. This tree can be coded as a number. I proved that there is a primitive recursive function that can check to see if a number represents a trace of the computation of substitution.

The key to solving this problem is to create a primitive recursive function that computes a bound on how large the code of the trace of computation can be for a given input. With this I created another primitive recursive function that searches for the trace of computation up to this bound. Once the trace is found—I proved that it must be found—the function extracts the result from the trace and returns it.

5.2 `checkPrf` Is Primitive Recursive

Given a code for a formula and a code for a proof, the function `checkPrf` returns 0 if the proof does not prove the formula, otherwise it returns one plus the code of the list of axioms used in the proof. I proved this function is primitive recursive, as well as proving that it is correct in the sense that for every proof p of φ from a list of axioms Γ, $\texttt{checkPrf}(\ulcorner\varphi\urcorner, \ulcorner p\urcorner) = 1 + \ulcorner\Gamma\urcorner$; and for all $n, m : \mathbb{N}$ if $\texttt{checkPrf}(n, m) \neq 0$ then there exists φ, Γ, and some proof p of φ from Γ such that $\ulcorner\varphi\urcorner = n$ and $\ulcorner p\urcorner = m$.

For any axiom system U expressible in T, I created the formulas `codeSysPrf` and `codeSysPf`. $\texttt{codeSysPrf}[\boldsymbol{x}_0/\ulcorner n\urcorner, \boldsymbol{x}_1/\ulcorner m\urcorner]$ is provable in T if m is the code of a proof in U of a formula coded by n. $\texttt{codeSysPf}[\boldsymbol{x}_0/\ulcorner n\urcorner]$ is provable in T if there exists a proof in U of a formula coded by n.

`codeSysPrf` and `codeSysPf` are not derived from a primitive recursive functions because I wanted to prove the incompleteness of axiom systems that may not have a primitive recursive characteristic function.

6 Fixed Point Theorem and Rosser's Incompleteness Theorem

The fixed point theorem states that for every formula φ there is some formula ψ such that

$$\mathrm{NN} \vdash \psi \Leftrightarrow \varphi[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$$

and that the free variables of ψ are that of φ less $\boldsymbol{x}_i$.

The fixed point theorem allows one to create "self-referential sentences". I used this to create Rosser's sentence which states that for every code of a proof of itself, there is a smaller code of a proof of its negation. The proof of Rosser's incompleteness theorem requires doing a bounded search for a proof, and this requires knowing what is and what is not a proof in the system. For this reason, I require the decidability of the axiom system. Without a decision procedure for the axiom system, I cannot constructively do the search.

6.1 Incompleteness of PA

To demonstrate the incompleteness theorem I used it to prove the incompleteness of PA. I created a primitive recursive predicate for the codes of the axioms of PA. Coq is sufficiently powerful to prove the consistency of PA by proving that the natural numbers model PA.

One subtle point is that Coq's logic is constructive while the internal logic is classical. One cannot interpret a formula of the internal logic directly in Coq and expect it to be provable if it is provable in the internal logic. Instead I use a double negation translation of the formulas. The translated formula will always hold if it holds in the internal logic.

The consistency of PA along with the expressibility of its axioms and the translations of proofs from NN to PA allowed me to apply Rosser's incompleteness theorem and prove the incompleteness of PA—there exists a sentence φ such that neither $\mathrm{PA} \vdash \varphi$ nor $\mathrm{PA} \vdash \neg\varphi$.

```
Theorem PAIncomplete :
 exists f : Formula,
   (forall v : nat, ~ In v (freeVarFormula LNT f)) /\
   ~ (SysPrf PA f \/ SysPrf PA (notH f)).
```

7 Remarks

7.1 Extracting the Sentence

Because my proof is constructive, it is possible, in principle, to compute this sentence that makes PA incomplete. This was not done for two reasons. The first reason is that the existential statement lives in Coq's `Prop` universe, and Coq's only extracts from its `Set` universe. This was an error on my part. I should have used Coq's `Set` existential quantifier; this problem would be fairly easy to fix. The second reason is that the sentence contains a closed term of the code

of most of itself. I believe this code is a very large number and it is written in unary notation. This would likely make the sentence far to large to be actually printed.

7.2 Robinson's System Q

The proof of essential incompleteness is usually carried out for Robinson's system Q. Instead I followed Hodel's development [10] and used NN. Q is PA with the induction schema replaced with $\forall \boldsymbol{x}_0.\exists \boldsymbol{x}_1.(\boldsymbol{x}_0 = \mathbf{0} \vee \boldsymbol{x}_0 = \boldsymbol{S}\boldsymbol{x}_1)$. All of NN axioms are Π_1 whereas Q has the above Π_2 axiom. Both axiom systems are finite.

Neither system is strictly weaker than the other, so it would not be possible to use the essential incompleteness of one to get the essential incompleteness of the other; however both NN and Q are sufficiently powerful to prove a small number of needed lemmas, and afterward only these lemmas are used. If one abstracts my proof at these lemmas, it would then be easy to prove the essential incompleteness of both Q and NN.

7.3 Comparisons with Shankar's 1986 Proof

It is worth noting the differences between this formalization of the incompleteness theorem and Shankar's 1986 proof in the Boyer-Moore theorem prover. The most notable difference is the proof systems. In Coq the user is expected to input the proof, in the form of a proof script, and Coq will check the correctness of the proof. In the Boyer-Moore theorem prover the user states a series of lemmas and the system generates the proofs. However, using the Boyer-Moore proof system requires feeding it a "well-chosen sequence of lemmas" [11, p. xii], so it would seem the information being fed into the two systems is similar.

There are some notable semantic differences between Shankar's statement of incompleteness and mine. His theorem only states that finite extensions of Z2, hereditarily finite set theory, are incomplete, whereas my theorem states that even infinite extensions of NN are incomplete as long as they are self-representable. Also Shankar's internal logic allows axioms to define new relation or function symbols as long as they come with the required proofs of admissibility. Such extensions are conservative over Z2, but no computer verified proof of this fact is given. My internal logic does not allow new symbols. Finally, I prove the essential incompleteness of NN, which is in the language of arithmetic. Without any set structures the proof is somewhat more difficult because it requires using Gödel's β-function.

One of Shankar's goals when creating his proof was to use a proof system without modifications. Unfortunately he was not able to meet that goal; he ended up making some improvements to the Boyer-Moore theorem prover. My proof was developed in Coq without any modifications.

7.4 Gödel's Second Incompleteness Theorem

The second incompleteness theorem states that if T is a recursive system extending PA—actually a weaker system could be used here—and $T \vdash \mathrm{Con}_T$ then T is

inconsistent. Con_T is some reasonable formula stating the consistency of T, such as $\neg \mathrm{Pr}_T(\ulcorner \mathbf{0} = \boldsymbol{S}\mathbf{0} \urcorner)$, where Pr_T is the provability predicate `codeSysPf` for T.

If I had created a formal proof in PA, I would have $\vdash_{\mathrm{PA}}$ "Gödel's first incompleteness theorem". This could then be mechanically transformed to create another formal proof in PA that $\vdash_{\mathrm{PA}}$ (PA $\vdash$ "Gödel's first incompleteness theorem"). The reader can verify that the second incompleteness theorem follows from this. Unfortunately I have only shown that $\vdash_{Coq}$ "Gödel's first incompleteness theorem", so the above argument cannot be used to create a proof of the second incompleteness theorem.

Still, this work can be used as a basis for formalizing the second incompleteness theorem. The approach would be to formalize the Hilbert-Bernays-Löb derivability conditions:

1. if PA $\vdash \varphi$ then PA $\vdash \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner)$
2. PA $\vdash \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner) \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner) \urcorner)$
3. PA $\vdash \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \Rightarrow \psi \urcorner) \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner) \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \psi \urcorner)$

The second condition is the most difficult to prove. It is usually proved by first proving that for every Σ_1 sentence φ, PA $\vdash \varphi \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner)$. Because I made sure that all primitive recursive functions are representable by a Σ_1 formula, it would be easy to go from this theorem to the second Hilbert-Bernays-Löb condition.

8 Statistics

My proof, excluding standard libraries and the library for Pocklington's criterion [2], consists of 46 source files, 7 036 lines of specifications, 37 906 lines of proof, and 1 267 747 total characters. The size of the gzipped tarball (`gzip -9`) of all the source files is 146 008 bytes, which is an estimate of the information content of my proof.

Acknowledgements

I would like to thank NSERC for providing funding for this research. I thank Robert Schneck for introducing me to Coq, and helping me out at the beginning. I would like to thank Nikita Borisov for letting me use his computer when the proof became to large for my poor laptop. I would also like to thank my Berkeley advisor, Leo Harrington, for his advice on improving upon Hodel's proof. And last, but not least, thanks to the Coq team, because without Coq there would be no proof.

References

1. Stanley N. Burris. Logic for mathematics and computer science: Supplementary text. `http://www.math.uwaterloo.ca/~snburris/htdocs/LOGIC/stext.html`, 1997.

2. Olga Caprotti and Martijn Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *J. Symb. Comput.*, 32(1/2):55–70, 2001.
3. Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in coq. In *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, London, UK, 1994. Springer-Verlag.
4. K. Gödel. Ueber Formal Unentscheidbare sätze der Principia Mathematica und Verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. english translation: On Formally Undecidable Propositions of Principia Mathematica and Related Systems I, Oliver & Boyd, London, 1962.
5. John Harrison. Formalizing basic first order model theory. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lecture Notes in Computer Science*, pages 153–170, Canberra, Australia, 1998. Springer-Verlag.
6. John Harrison. The HOL-Light manual, 2000.
7. Claude Marche. Fwd: Question about fixpoint. Coq club mailing list correspondence, `http://pauillac.inria.fr/pipermail/coq-club/2005/001641.html`, [cited 2005-02-07], February 2005.
8. Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.
9. Russell O'Connor. The Gödel-Rosser 1st incompleteness theorem. `http://r6.ca/Goedel20050512.tar.gz`, March 2005.
10. Richard E. Hodel. *An Introduction to Mathematical Logic.* PWS Pub. Co., 1995.
11. N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1994.
12. J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.
13. Allen Stoughton. Substitution revisited. 59(3):317–325, August 1988.
14. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.