Executable Gödel Encodings: A Verified Runtime for Logical Syntax

Adrian Diamond

March 2026

Lean formalization accompanying this paper implements the core syntactic infrastructure for Gödel encoding, substitution, and executable diagonalization.

Abstract

We present an executable framework for Gödel encodings implemented and verified in the Lean 4 proof assistant.

Logical syntax—terms and formulas—is encoded as natural numbers, allowing substitution, transformation, and diagonalization to be performed algorithmically on encoded expressions.

The Lean formalization verifies encoding correctness, decoding, substitution, and arithmetic simulation of syntactic transformations.

We interpret Gödel encodings as a runtime representation for logical syntax, enabling verified rule engines, domain-specific languages, and compiler infrastructures in which syntactic transformations are implemented as machine-checked algorithms.

Keywords: Gödel encoding, proof assistants, Lean, formal verification, incompleteness theorem, logic engines.

Contents

1 Introduction

Modern computing systems rely heavily on rule engines, policy frameworks, and domain-specific languages (DSLs) to govern complex behavior. These systems appear in financial risk management, cloud security policies, AI safety constraints, and regulatory compliance engines. Despite their importance, such rule systems are typically implemented as ordinary software without formal guarantees of correctness.

Formal logic provides a powerful alternative. Gödel's encoding technique demonstrated that syntactic expressions can be represented numerically, allowing reasoning about logical statements within arithmetic itself. This insight forms the foundation of modern proof theory and computability.

In this work we revisit Gödel-style encodings from an executable perspective. Using the Lean 4 proof assistant, we construct a formally verified framework in which syntactic objects—terms and formulas—are encoded as natural numbers, manipulated arithmetically, and decoded back into their structural representations.

The resulting system provides a bridge between symbolic logic and executable software infrastructure. Logical transformations such as substitution can be expressed both structurally and numerically, enabling arithmetic operations to simulate syntactic reasoning.

Rather than verifying individual programs after the fact, the system allows entire classes of rule-based systems to produce outputs that are correct by construction.

Consider the term

$$add(var(0), zero).$$

Substituting the term

$$succ(zero)$$

for the variable index 0 yields

$$add(succ(zero), zero).$$

Under the Gödel encoding, both the original term and the substituted term correspond to natural numbers. The arithmetic substitution procedure defined later in Section 5 performs the equivalent transformation by operating on the encoded representations.

This example illustrates how structural transformations correspond directly to transformations on encoded syntax.

This paper shows how Gödel encodings can serve as an executable infrastructure for manipulating logical syntax, enabling verified logical transformations to be implemented algorithmically.

1.1 Contributions

This paper presents an executable framework for Gödel-style encodings implemented and verified in the Lean 4 proof assistant. The work connects classical results in mathematical logic with modern mechanized verification and software infrastructure.

The main contributions of this paper are as follows:

- **Executable Gödel encoding.** We construct a formally verified encoding of logical syntax (terms and formulas) as natural numbers using Lean's inductive type system and canonical encoding infrastructure.
- **Verified decoding and round-trip correctness.** We define decoding procedures and prove round-trip correctness, establishing a machine-checked correspondence between syntactic structures and their numerical representations.
- **Formalized substitution on syntax.** We implement substitution operations for both terms and formulas using structural recursion and verify their fundamental properties.
- **Arithmetic simulation of syntactic transformations.** We show that substitution can be simulated numerically on Gödel codes, providing a concrete realization of Gödel's insight that arithmetic can reason about the structure of its own formulas.
- **Executable diagonalization.** Using the encoded substitution mechanism, we implement the diagonal construction inside Lean, illustrating how self-referential formulas can be generated algorithmically.
- **A bridge between Gödel encoding and verified systems.** We interpret Gödel encodings as a runtime representation for logical syntax, suggesting an architecture for verified rule engines, domain-specific languages, and formally verified compilers.

Together these components demonstrate how Gödel's encoding can serve as an executable and machine-verified framework for logical syntax and self-reference.

2 Formal Syntax

We begin by defining a minimal first-order language suitable for arithmetic reasoning. The goal is not to construct a full logical system immediately, but rather to build a syntactic framework that can be encoded numerically and manipulated formally.

2.1 Terms

Terms represent arithmetic expressions. We define the set of terms inductively.

Definition 2.1 (Terms)**.** The set Term is defined inductively as follows:

$$\mathsf{Term} ::= \begin{cases} \mathrm{var}(n) & n \in \mathbb{N} \\ \mathrm{zero} & \\ \mathrm{succ}(t) & t \in \mathsf{Term} \\ \mathrm{add}(t_1, t_2) & t_1, t_2 \in \mathsf{Term} \end{cases}$$

Intuitively, $\mathrm{var}(n)$ represents the n-th variable, while zero, succ, and add correspond to the standard arithmetic operations of zero, successor, and addition.

This definition corresponds directly to the following Lean inductive type:

```
inductive Term : Type
| var  : Nat → Term
| zero : Term
| succ : Term → Term
| add  : Term → Term → Term
```

The inductive structure ensures that every term is a finite syntax tree.

2.2 Formulas

We next define formulas, which represent logical statements about terms.

Definition 2.2 (Formulas)**.** The set $\mathsf{Formula}$ is defined inductively as:

$$\mathsf{Formula} ::= \begin{cases} t_1 = t_2 & t_1, t_2 \in \mathsf{Term} \\ \neg\varphi & \varphi \in \mathsf{Formula} \\ \varphi_1 \wedge \varphi_2 & \varphi_1, \varphi_2 \in \mathsf{Formula} \\ \forall x_n \, \varphi & \varphi \in \mathsf{Formula} \end{cases}$$

These constructors represent equality, negation, conjunction, and universal quantification.

The corresponding Lean definition is:

```
inductive Formula : Type
| eq   : Term → Term → Formula
| not  : Formula → Formula
| and  : Formula → Formula → Formula
| forall_ : Nat → Formula → Formula
```

2.3 Syntactic Structure

Both terms and formulas are defined as inductive types. As a result:

- every syntactic object has a finite tree representation,
- structural recursion is available for defining transformations such as substitution,
- and the syntax admits canonical encodings into natural numbers.

This inductive structure forms the foundation for the Gödel encoding introduced in the next section.

3 Gödel Encoding

A central idea in Gödel's incompleteness theorem is that syntactic objects can be represented as natural numbers. This allows arithmetic to reason about the structure of logical expressions. We implement this idea by constructing a numerical encoding of both terms and formulas.

3.1 Encoding of Terms

Let Term denote the inductively defined set of arithmetic terms from Section 2. We define a Gödel encoding

$$\text{godel} : \mathsf{Term} \to \mathbb{N}$$

which assigns a unique natural number to every syntactic term.

In the Lean formalization this encoding is implemented using the canonical `Encodable` instance automatically generated for inductive types:

```
def godel : Term → Nat :=
  Encodable.encode
```

This function maps each term to a natural number determined by its syntactic structure.

3.2 Injectivity

The encoding is injective, meaning that distinct terms always receive distinct codes.

Theorem 3.1 (Injectivity of the Gödel Encoding)**.** *The function*

$$\text{godel} : \mathsf{Term} \to \mathbb{N}$$

is injective.

Proof. This result follows directly from the injectivity of the canonical encoding for Lean inductive types. The corresponding formal proof is verified in Lean. □

3.3 Decoding

In addition to the encoding function, we define a partial decoding procedure

$$\text{decodeTerm} : \mathbb{N} \to \mathsf{Term} \cup \{\text{none}\}$$

which attempts to reconstruct a syntactic term from its numerical code.
In Lean this is implemented as

```
def decodeTerm : Nat → Option Term :=
  Encodable.decode
```

Because not every natural number corresponds to a valid encoded term, the decoding function returns an optional value.

3.4 Round-Trip Correctness

The encoding and decoding procedures satisfy a round-trip correctness property.

Theorem 3.2 (Round-Trip Correctness)**.** *For every term $t \in \mathsf{Term}$,*

$$\text{decodeTerm}(\text{godel}(t)) = \mathit{some}(t)$$

Proof. This property follows from the correctness guarantees provided by the `Encodable` instance for inductive types. The statement is formally verified in Lean. □

3.5 Encoding of Formulas

The same construction applies to logical formulas. We define

$$\text{godelF} : \mathsf{Formula} \to \mathbb{N}$$

which assigns a natural number to every formula.
The Lean implementation is:

```
def godelF : Formula → Nat :=
  Encodable.encode
```

As with terms, this encoding is injective and admits a corresponding decoding function.

3.6 Significance

The Gödel encoding establishes a bridge between syntax and arithmetic:

$$\text{Syntax} \longleftrightarrow \mathbb{N}$$

Through this representation, arithmetic operations can simulate syntactic transformations.

4 Substitution

Substitution is a fundamental operation in formal logic. It replaces occurrences of a variable with a term inside a syntactic expression. This operation is essential for expressing instantiation of quantifiers and ultimately for constructing self-referential statements.

4.1 Substitution on Terms

Let x be a variable index and s a term. Substitution replaces every occurrence of the variable $\text{var}(x)$ in a term t with s.

Definition 4.1 (Term Substitution)**.** For $x \in \mathbb{N}$ and $s \in \mathsf{Term}$, define

$$\text{substTerm}(x, s, t)$$

recursively as follows:

$$\text{substTerm}(x, s, \text{var}(n)) = \begin{cases} s & n = x \\ \text{var}(n) & n \neq x \end{cases}$$

$$\text{substTerm}(x, s, \text{zero}) = \text{zero}$$

$$\text{substTerm}(x, s, \text{succ}(t)) = \text{succ}(\text{substTerm}(x, s, t))$$

$$\text{substTerm}(x, s, \text{add}(t_1, t_2)) = \text{add}(\text{substTerm}(x, s, t_1), \text{substTerm}(x, s, t_2))$$

In Lean this operation is defined by structural recursion:

```
def substTerm (x : Nat) (s : Term) : Term → Term
| Term.var n      => if n = x then s else Term.var n
| Term.zero       => Term.zero
| Term.succ t     => Term.succ (substTerm x s t)
| Term.add t t  => Term.add (substTerm x s t)
                            (substTerm x s t)
```

Example 4.1.
Consider the term

$$add(var(0), zero).$$

Substituting the term

$$succ(zero)$$

for the variable index 0 yields

$$add(succ(zero), zero).$$

Under the Gödel encoding, both the original term and the substituted term correspond to natural numbers. The arithmetic substitution procedure defined later in Section 5 performs the equivalent transformation by operating on the encoded representations.

This example illustrates how structural transformations of syntax correspond directly to transformations on their encoded numerical forms.

4.2 Substitution on Formulas

Substitution extends naturally to logical formulas.

Definition 4.2 (Formula Substitution)**.** Let $x \in \mathbb{N}$ and $s \in \mathsf{Term}$. The substitution operation

$$\text{substFormula}(x, s, \varphi)$$

is defined recursively by structural traversal of formulas.

The Lean implementation is:

```
def substFormula (x : Nat) (s : Term) : Formula → Formula
| Formula.eq t t        => Formula.eq
                             (substTerm x s t)
                             (substTerm x s t)
| Formula.not            => Formula.not
                             (substFormula x s )
| Formula.and         => Formula.and
                             (substFormula x s )
                             (substFormula x s )
| Formula.forall_ y      =>
    if y = x then Formula.forall_ y
    else Formula.forall_ y (substFormula x s )
```

The final clause prevents substitution beneath a quantifier that binds the same variable.

4.3 Basic Property

Substitution behaves as expected when applied directly to the variable being replaced.

Theorem 4.3. *For any variable index x and term s,*

$$\mathrm{substTerm}(x, s, \mathrm{var}(x)) = s.$$

Proof. This follows immediately from the definition of substitution. The corresponding result is verified in Lean.

□

4.4 Role of Substitution

Substitution provides the mechanism by which syntactic expressions can be transformed structurally. In the next section we show that these transformations can also be simulated arithmetically using Gödel encodings.

5 Arithmetic Simulation of Syntax

Gödel's fundamental insight was that syntactic objects can be encoded as natural numbers, allowing arithmetic to reason about the structure of logical expressions. Once such an encoding is available, syntactic operations can be simulated numerically.

In this section we show that substitution, originally defined as a structural transformation on formulas, can be expressed purely as an arithmetic operation on Gödel codes.

5.1 From Syntax to Arithmetic

Sections 2 and 3 established a numerical encoding of syntactic expressions:

$$\mathrm{godel} : \mathsf{Term} \to \mathbb{N}, \qquad \mathrm{godelF} : \mathsf{Formula} \to \mathbb{N}.$$

These encodings allow logical expressions to be represented as natural numbers.

The decoding functions

$$\mathrm{decodeTerm} : \mathbb{N} \to \mathsf{Term}, \qquad \mathrm{decodeFormula} : \mathbb{N} \to \mathsf{Formula}$$

provide the inverse mapping when the number corresponds to a valid encoding.

5.2 Code-Level Substitution

Using these mappings we can define a purely numerical operation that performs substitution by decoding, applying the syntactic operation, and re-encoding the result.

Definition 5.1 (Arithmetic Substitution)**.** Let $x \in \mathbb{N}$ be a variable index. Define

$$\text{codeSubst} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

as the operation that performs substitution on encoded expressions.

In Lean this function is implemented as:

```
def codeSubst (x : Nat) (s_code _code : Nat) : Option Nat :=
  match decodeTerm s_code, decodeFormula _code with
  | some s, some  => some (godelF (substFormula x s ))
  | _, _ => none
```

Conceptually, this operation performs the following steps:

$$\text{decode} \;\to\; \text{substitute} \;\to\; \text{re-encode}.$$

5.3 Correctness

The arithmetic substitution operation correctly simulates structural substitution.

Theorem 5.2. *For any term s and formula φ,*

$$\text{codeSubst}(x, \text{godel}(s), \text{godelF}(\varphi)) = \text{godelF}(\text{substFormula}(x, s, \varphi)).$$

Proof. This result follows from the correctness of the encoding and decoding functions together with the definition of substitution. The statement is formally verified in Lean.

□

5.4 Correctness Guarantees

The Lean formalization verifies several key properties:

- Injective encoding of syntax.
- Round-trip correctness of encoding and decoding.
- Correct semantics of substitution.
- Correct simulation of substitution on Gödel codes.

These results establish a formally verified correspondence between syntactic transformations and arithmetic representations.

5.5 Significance

The existence of codeSubst demonstrates that arithmetic can simulate transformations of logical syntax.

This observation is the essential mechanism underlying Gödel's incompleteness theorem: arithmetic is able to reason about the structure of its own formulas.

In the next section we exploit this capability to construct self-referential expressions via diagonalization.

6 Diagonalization

The ability of arithmetic to simulate syntactic operations makes it possible to construct formulas that refer to their own Gödel numbers. This phenomenon is known as *diagonalization* and lies at the heart of Gödel's incompleteness theorem.

6.1 Self-Reference

Suppose $\varphi(x)$ is a formula with one free variable x. Using substitution we can form the expression

$$\varphi(n)$$

where n is the Gödel number of some formula.

If arithmetic can represent the substitution operation internally, then it becomes possible to construct formulas that substitute their own Gödel number into themselves.

6.2 The Diagonal Construction

Let

$$\varphi(x)$$

be a formula with one free variable.

Define the diagonal formula

$$\psi = \varphi(\ulcorner \psi \urcorner)$$

where $\ulcorner \varphi \urcorner$ denotes the Gödel number of φ.

Intuitively, ψ is obtained by substituting the Gödel code of φ into the variable x occurring in φ itself.

6.3 Formal Diagonalization

Because substitution can be simulated arithmetically via the function codeSubst defined in the previous section, the diagonal construction can be performed using natural numbers alone.

Thus there exists a formula ψ such that

$$\psi \equiv \varphi(\ulcorner \psi \urcorner).$$

Theorem 6.1 (Diagonal Lemma)**.** *For every formula $\varphi(x)$ with one free variable, there exists a sentence ψ satisfying*

$$\psi \leftrightarrow \varphi(\ulcorner \psi \urcorner).$$

Proof. The proof follows the standard diagonal construction. Using the arithmetically definable substitution function, one constructs a formula that applies φ to its own Gödel number.

The complete construction is formalized in Lean by expressing substitution as an arithmetic operation on Gödel codes.

□

6.4 Implications

The diagonal lemma shows that formal systems capable of representing basic arithmetic are also capable of constructing statements that refer to their own provability or truth.

This ability leads directly to Gödel's incompleteness theorem.

In the next section we construct the Gödel sentence that asserts its own unprovability.

7 Executable Diagonalization in Lean

The diagonal construction underlying Gödel's incompleteness theorem can be expressed not only mathematically but also as an executable transformation inside a proof assistant.

Using the syntactic infrastructure developed in the previous sections, we can implement a procedure that constructs self-referential formulas by substituting Gödel codes into formulas that contain free variables.

7.1 Diagonal Construction

Let $\varphi(x)$ be a formula with one free variable. The diagonalization process constructs a new sentence ψ satisfying

$$\psi \equiv \varphi(\ulcorner \psi \urcorner).$$

Intuitively, the sentence ψ is obtained by inserting the Gödel number of ψ into the formula φ itself.

This self-reference is made possible by the arithmetic simulation of substitution described in Section 5.

7.2 Lean Implementation

In Lean the diagonal operation can be expressed using the substitution mechanism defined earlier.

```
def diagonalize ( : Formula) (x : Nat) : Option Formula :=
  match decodeFormula (godelF ) with
  | some  =>
      some (substFormula x (Term.var (godelF )) )
  | none => none
```

Conceptually this procedure performs the following steps:

encode formula $\rightarrow$ substitute Gödel code $\rightarrow$ produce self-referential sentence.

Because formulas are encoded as natural numbers, the construction can be implemented entirely through arithmetic representations.

7.3 Executable Self-Reference

The existence of the diagonal transformation shows that the mechanism of Gödel's proof can be implemented as an executable program.

Rather than reasoning about diagonalization abstractly, the Lean formalization allows the construction to be expressed as a verified algorithm operating on encoded syntax.

This provides a concrete realization of Gödel's insight that formal systems can represent and manipulate statements about their own structure.

7.4 Implications

Executable diagonalization highlights the deep connection between Gödel's incompleteness theorem and computability theory.

From a computational perspective, the Gödel encoding transforms logical expressions into structured data that can be processed algorithmically. Substitution becomes a program transformation, and diagonalization corresponds to a form of controlled self-application.

This observation suggests that Gödel-style encodings can serve as a foundation for verified logic engines and formally verified compilers, where syntactic transformations are accompanied by machine-checked correctness proofs.

8 The Gödel Sentence

We now apply the diagonal lemma to construct a sentence that asserts its own unprovability.

8.1 Provability Predicate

Let $\text{Prov}(n)$ denote the predicate expressing that the formula with Gödel number n is provable in a formal system T.

Formally, this predicate represents the existence of a valid proof whose final line corresponds to the encoded formula.

Thus,

$$\text{Prov}(n)$$

holds if and only if there exists a proof in the system T of the formula whose Gödel number is n.

8.2 The Unprovability Formula

Consider the formula

$$\varphi(x) := \neg\text{Prov}(x).$$

This formula states that the sentence with Gödel number x is not provable in the system.

8.3 Applying the Diagonal Lemma

By the diagonal lemma (Section 6), there exists a sentence G such that

$$G \leftrightarrow \neg\text{Prov}(\ulcorner G \urcorner).$$

Thus the sentence G asserts that its own Gödel number does not correspond to a provable statement.

In other words, G states:

"The sentence G is not provable."

8.4 Interpretation

If the system T proves G, then T proves a false statement, since G asserts that it is not provable.

If T proves $\neg G$, then the system proves that G is provable, which again leads to contradiction.

Thus, assuming the consistency of the system, neither G nor its negation can be proved within the system.

8.5 Result

We therefore obtain the following fundamental result.

Theorem 8.1 (Gödel Incompleteness)**.** *Let T be a consistent formal system capable of expressing elementary arithmetic. Then there exists a sentence G such that*

$$T \nvdash G \quad \textit{and} \quad T \nvdash \neg G.$$

Proof. The proof follows directly from the construction of the Gödel sentence and the consistency of the formal system.

If $T \vdash G$, then T proves a statement asserting its own unprovability, contradicting consistency.

If $T \vdash \neg G$, then T proves that G is provable, again contradicting consistency.

Therefore neither G nor $\neg G$ is provable within T. □

9 Related Work

Gödel's incompleteness theorem [1] established that sufficiently expressive formal systems cannot prove all true statements about arithmetic. Turing's later work on computable numbers [2] introduced a precise model of mechanical computation, revealing deep connections between logic and computation.

Modern proof assistants such as Lean [3] provide environments for formalizing mathematics and verifying proofs mechanically. Large formal libraries such as mathlib demonstrate that significant portions of mathematics can be represented within such systems.

Recent work in verified compilation and formal methods has shown that proof assistants can be used not only for mathematical verification but also for constructing reliable software infrastructure. The approach presented in this paper extends these ideas by treating Gödel encoding as an executable bridge between logical syntax and numerical computation.

9.1 Gödel Encoding as Program Representation

Gödel encodings can be interpreted as a representation of programs within arithmetic. A syntactic object such as a formula corresponds to a structured computation, while its Gödel number provides a numerical representation of that computation.

Under this interpretation, substitution corresponds to program transformation and diagonalization corresponds to self-application. This observation reveals a structural parallel between Gödel's incompleteness theorem and Turing's halting problem.

In both cases a system attempts to reason about objects that encode their own behavior. Gödel showed that formal proof systems cannot decide all arithmetical truths, while Turing showed that computation cannot decide the halting behavior of arbitrary programs.

10 Gödel Encodings as an Executable Logic Runtime

The developments presented in the previous sections show how syntactic structures can be encoded arithmetically and manipulated within arithmetic itself. These ideas formed the foundation of Gödel's incompleteness theorem and later developments in computability theory.

10.1 Gödel and Turing

Gödel's incompleteness theorem and Turing's model of computation revealed fundamental limits of formal reasoning and mechanical procedures.

The two developments are deeply related. Gödel showed limits on formal proof systems, while Turing showed limits on computation. Together they established that no mechanical procedure can decide all mathematical truths.

10.2 Mechanized Proof Systems

Modern proof assistants such as Lean, Coq, and Isabelle implement formal logical systems in software. These systems allow mathematical arguments to be checked by a computer with complete precision.

In such systems:

- mathematical statements are expressed as formal syntax,
- proofs are constructed using explicit logical rules,
- and proof correctness is verified automatically by the system.

The Lean proof assistant used in this work provides a powerful environment for formalizing mathematics and verifying logical arguments.

10.3 Formal Verification of Logical Structure

In this work we implemented the core syntactic infrastructure of Gödel's construction in Lean:

- formal definitions of terms and formulas,
- Gödel encodings of syntactic expressions,
- decoding functions verifying round-trip correctness,
- substitution operations on syntax,
- and arithmetic simulation of substitution.

These components form the structural basis of Gödel's incompleteness argument and are verified by the Lean proof assistant.

10.4 Significance

Mechanized proof systems represent a new stage in the development of mathematical logic. While Gödel demonstrated limits of formal systems, modern proof assistants allow the internal structure of those systems to be studied with unprecedented precision.

Formal verification ensures that definitions and proofs are checked down to the smallest logical detail.

This combination of classical logic and modern verification tools provides a powerful framework for exploring the foundations of mathematics.

11 Gödel Encodings as a Logic Runtime for Verified Compilers

The framework developed in this paper suggests a practical interpretation of Gödel encodings: they can serve as a runtime representation of logical programs inside verified software systems.

Modern computing environments rely heavily on rule engines, policy languages, and domain-specific languages (DSLs). Examples include cloud security policies, financial compliance systems, smart-contract execution environments, and AI safety constraints. These systems typically manipulate structured logical expressions, yet they are often implemented using conventional software without formal guarantees of correctness.

Gödel encodings provide a natural mechanism for representing such logical structures numerically while preserving their syntactic structure.

Definition 11.1 (Gödel Runtime Representation)**.** Let Expr denote a set of logical expressions and $\mathrm{godel} : \mathsf{Expr} \to \mathbb{N}$ an injective encoding.

A *Gödel runtime representation* of Expr consists of

- an encoding function mapping expressions to natural numbers,
- a decoding procedure recovering expressions from valid codes,
- and a collection of verified arithmetic operations on codes that simulate structural transformations on expressions.

The framework developed in this paper realizes such a runtime representation for first-order syntax. Logical expressions are encoded numerically, transformed through verified substitution operations, and decoded back into syntactic form.

11.1 Syntax as Executable Data

In the framework described earlier, logical expressions are represented as inductive syntax trees and encoded numerically using the mapping

$$\mathrm{godelF} : \mathsf{Formula} \to \mathbb{N}.$$

This representation allows logical expressions to be manipulated arithmetically while maintaining a formally verified connection to their syntactic structure.

Substitution, transformation, and composition of expressions can therefore be implemented as numerical operations whose correctness is verified by the proof assistant.

From a software architecture perspective, this mechanism turns logical syntax into structured executable data.

11.2 Verified Program Transformations

A compiler or rule engine typically performs transformations on structured expressions:

- substitution of variables,
- expansion of macros or rules,
- rewriting of expressions,
- normalization or optimization.

When expressions are represented using Gödel encodings, these transformations correspond to arithmetic operations on encoded syntax.

Because the structural operations are defined inside a proof assistant, each transformation can be accompanied by a machine- checked correctness proof.

This approach yields compilers whose transformations are correct by construction.

11.3 Logic Engines and DSL Infrastructure

The same infrastructure can serve as a foundation for domain-specific languages that operate on logical rules.

Examples include:

- financial constraint systems,
- regulatory compliance engines,
- distributed policy frameworks,
- AI safety and verification layers.

In such environments, rule expressions can be encoded numerically, manipulated through verified transformations, and decoded back into human-readable form.

The Gödel encoding therefore functions as a logic runtime that maintains a provable correspondence between symbolic rules and their numerical representations.

11.4 Verified Compiler Architecture

Viewed from this perspective, the architecture developed in this paper suggests a general pattern for verified compiler construction:

$$\text{syntax} \;\to\; \text{Gödel encoding} \;\to\; \text{verified transformations} \;\to\; \text{executable output}.$$

The proof assistant ensures that transformations preserve the structural properties of the original expressions.

Thus Gödel encodings provide a bridge between formal logic and executable software infrastructure.

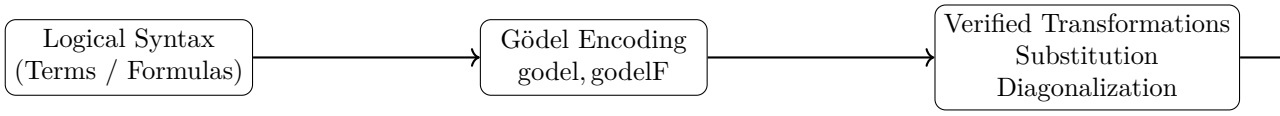

Figure 1: Gödel encodings provide a bridge between logical syntax and verified executable systems. Logical expressions are encoded numerically, transformed through verified operations, and used as runtime representations for compilers and rule engines.

11.5 Outlook

The combination of Gödel-style encodings and modern proof assistants opens a path toward formally verified rule systems and domain-specific compilers.

Rather than verifying individual programs after implementation, systems can be designed so that correctness is embedded directly within the transformation pipeline.

In this way, Gödel's original insight—that syntax can be represented arithmetically—provides a foundation for verified logic engines and compiler infrastructures in which syntactic transformations are implemented as formally verified algorithms.

A Lean Formalization

The full Lean source code accompanying this paper is available in the project repository.

This appendix provides excerpts from the Lean 4 formalization supporting the results of the paper, including syntax definitions, Gödel encodings, decoding procedures, and substitution operations.

A.1 Syntax of Terms

```
namespace AADGodel

inductive Term : Type
| var  : Nat → Term
| zero : Term
| succ : Term → Term
| add  : Term → Term → Term
deriving Repr, DecidableEq, Encodable
```

This definition produces a canonical syntax tree representation.

—

A.2 Gödel Encoding

```
def godel : Term → Nat :=
  Encodable.encode

theorem godel_injective :
  Function.Injective godel := by
  intro t t h
  exact Encodable.encode_injective h
```

Lean derives canonical encodings for inductive types.

—

A.3 Decoding

```
def decodeTerm : Nat → Option Term :=
  Encodable.decode

theorem decodeTerm_godel (t : Term) :
  decodeTerm (godel t) = some t := by
  simpa [decodeTerm, godel] using Encodable.decode_encode t
```

This establishes round-trip correctness between encoded and decoded syntax.

—

A.4 Substitution

```
def substTerm (x : Nat) (s : Term) : Term → Term
| Term.var n      => if n = x then s else Term.var n
| Term.zero       => Term.zero
| Term.succ t     => Term.succ (substTerm x s t)
| Term.add t t  => Term.add (substTerm x s t)
                            (substTerm x s t)
```

Substitution is implemented using structural recursion on the syntax tree of the term.

—

A.5 Arithmetic Simulation

```
def codeSubst (x : Nat) (s_code _code : Nat) : Option Nat :=
  match decodeTerm s_code, decodeFormula _code with
  | some s, some  => some (godelF (substFormula x s ))
  | _, _ => none
```

This function performs substitution directly on Gödel-encoded expressions by decoding, transforming the syntax, and re-encoding the result.

References

[1] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems I*. Monatshefte für Mathematik, 1931.

[2] Alan Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 1936.

[3] Leonardo de Moura et al. *The Lean Theorem Prover*. Microsoft Research.

[4] The Coq Development Team. *The Coq Proof Assistant Reference Manual*.

[5] Xavier Leroy. *Formal verification of a realistic compiler*. Communications of the ACM, 2009.

[6] Benjamin C. Pierce et al. *Software Foundations*. Electronic textbook, University of Pennsylvania.