Löb's Theorem

A functional pearl of dependently typed quining

Anonymous

Keywords Agda, Löb's theorem, quine, self-reference, type theory

Abstract

Löb's theorem states that to prove that a proposition is provable, it is sufficient to prove the proposition under the assumption that it is provable. The Curry-Howard isomorphism identifies formal proofs with abstract syntax trees of programs; Löb's theorem thus implies, for total languages which validate it, that self-interpreters are impossible. We formalize a few variations of Löb's theorem in Agda using an inductive-inductive encoding of terms indexed over types. We verify the consistency of our formalizations relative to Agda by giving them semantics via interpretation functions.

1. Introduction

> *If P's answer is 'Bad!', Q will suddenly stop.*
> *But otherwise, Q will go back to the top,*
> *and start off again, looping endlessly back,*
> *till the universe dies and turns frozen and black.*
>
> Excerpt from *Scooping the Loop Snooper: A proof that the Halting Problem is undecidable* (Pullum 2000)

Löb's theorem has a variety of applications, from providing an induction rule for program semantics involving a "later" operator (Appel et al. 2007), to proving incompleteness of a logical theory as a trivial corollary, from acting as a no-go theorem for a large class of self-interpreters, to allowing robust cooperation in the Prisoner's Dilemma with Source Code (Barasz et al. 2014), and even in one case curing social anxiety (Yudkowsky 2014).

In this paper, after introducing the content of Löb's theorem, we will present in Agda three formalizations of type-theoretic languages and prove Löb's theorem in and about these languages: one that shows the theorem is admissible as an axiom in a wide range of situations, one which proves Löb's theorem by assuming as an axiom the existence of quines (programs which output their own source code), and one which constructs the proof under even weaker assumptions; see section 5 for details.

"What is Löb's theorem, this versatile tool with wondrous applications?" you may ask.

Consider the sentence "if this sentence is true, then you, dear reader, are the most awesome person in the world." Suppose that this sentence is true. Then you are the most awesome person in the world. Since this is exactly what the sentence asserts, the sentence is true, and you are the most awesome person in the world. For those more comfortable with symbolic logic, we can let X be the statement "you, dear reader, are the most awesome person in the world", and we can let A be the statement "if this sentence is true, then X". Since we have that A and $A \to X$ are the same, if we assume A, we are also assuming $A \to X$, and hence we have X. Thus since assuming A yields X, we have that $A \to X$. What went wrong?[1]

It can be made quite clear that something is wrong; the more common form of this sentence is used to prove the existence of Santa Claus to logical children: considering the sentence "if this sentence is true, then Santa Claus exists", we can prove that Santa Claus exists. By the same logic, though, we can prove that Santa Claus does not exist by considering the sentence "if this sentence is true, then Santa Claus does not exist." Whether you consider it absurd that Santa Claus exist, or absurd that Santa Claus not exist, surely you will consider it absurd that Santa Claus both exist and not exist. This is known as Curry's Paradox.

The problem is that the phrase "this sentence is true" is not a valid mathematical assertion; no language can encode a truth predicate for itself (Tarski 1936). However, some languages *can* encode assertions about *provability* (Gödel 1931). In section 2, we will dig into the difference between truth and provability from a computational prospective. We will present an argument for the indefinability of truth and for the definability of provability, which we hope will prove enlightening when we get to the formalization of Löb's theorem itself.

Now consider the sentence "if this sentence is provable, then Santa Claus exists." Fix a formal system powerful enough to talk about which of its sentences are provable (for example, Peano Arithmetic, Martin–Löf Type Theory, or Gödel-Löb Modal Logic), and fix a formalization of provability in that system. To prove that our sentence is true, we suppose that it is provable. We must now show that Santa Claus exists. *If provability implies truth*, then the sentence is true, and thus Santa Claus exists. Hence, if we can assume that provability implies truth, then we can prove that the sentence is true. This, in a nutshell, is Löb's theorem: to prove X, it suffices to prove that X is true whenever X is provable. If we let $\Box X$ denote the assertion "X is provable," then, symbolically, Löb's theorem becomes:

$$\Box(\Box X \to X) \to \Box X.$$

2. Quines and the Curry–Howard Isomorphism

Let us now return to the question we posed above: what went wrong with our original sentence? The answer is that self-reference with

[1] Those unfamiliar with conditionals should note that the "if … then …" we use here is the logical "if", where "if false then X" is always true, and not the counter-factual "if".

Logic	Programming	Set Theory
Proposition	Type	Set of Proofs
Proof	Program	Element
Implication ($\to$)	Function ($\to$)	Function
Conjunction ($\wedge$)	Pairing (,)	Cartesian Product ($\times$)
Disjunction ($\vee$)	Sum (+)	Disjoint Union ($\sqcup$)
Gödel codes	ASTs	—
$\Box\, X \to X$	Interpreters	—
(In)completeness	Halting problem	—

Table 1. The Curry-Howard Isomorphism between mathematical logic and functional programming

truth is impossible, and the clearest way we know to argue for this is via the Curry–Howard Isomorphism; in a particular technical sense, the problem is that self-reference with truth fails to terminate.

The Curry–Howard Isomorphism establishes an equivalence between types and propositions, between (well-typed, terminating, functional) programs and proofs. See Table 1 for some examples. Now we ask: what corresponds to a formalization of provability? If a proof of P is a terminating functional program which is well-typed at the type corresponding to P, and to assert that P is provable is to assert that the type corresponding to P is inhabited, then an encoding of a proof is an encoding of a program. Although mathematicians typically use Gödel codes to encode propositions and proofs, a more natural choice of encoding programs is abstract syntax trees (ASTs). In particular, a valid syntactic proof of a given (syntactic) proposition corresponds to a well-typed syntax tree for an inhabitant of the corresponding syntactic type.

Note well that the type ($\Box\, X \to X$) is a type that takes syntax trees and evaluates them; it is the type of an interpreter or an unquoter.

Unless otherwise specified, we will henceforth consider only well-typed, terminating programs; when we say "program", the adjectives "well-typed" and "terminating" are implied.

What is the computational equivalent of the sentence "If this sentence is provable, then X"? It will be something of the form "??? $\to X$". As a warm-up, let's look at a Python program that outputs its own source code.

There are three genuinely distinct solutions, the first of which is degenerate, and the second of which is cheeky. These solutions are:

- The empty program, which outputs nothing.
- The code `print(open(__file__, 'r').read())`, which relies on the Python interpreter to get the source code of the program.
- A program with a "template" which contains a copy of the source code of all of the program except for the template itself, leaving a hole where the template should be. The program then substitutes a quoted copy of the template into the hole in the template itself. In code, we can use Python's `repr` to get a quoted copy of the template, and we do substitution using Python's replacement syntax: for example, `("foo %s bar" % "baz")` becomes `"foo baz bar"`. Our third solution, in code, is thus:

  ```
  T = 'T = %s\nprint(T %% repr(T))'
  print(T % repr(T))
  ```

  The functional equivalent, which does not use assignment, and which we will be using later on in this paper, is:

  ```
  (lambda T: T % repr(T))
   ('(lambda T: T %% repr(T))\n (%s)')
  ```

We can use this technique, known as quining (Hofstadter 1979; Kleene 1952), to describe self-referential programs.

Suppose Python had a function $\Box$ that took a quoted representation of a Martin–Löf type (as a Python string), and returned a Python object representing the Martin–Löf type of ASTs of Martin–Löf programs inhabiting that type. Now consider the program

```
φ = (lambda T: □(T % repr(T)))
     ('(lambda T: □(T %% repr(T)))\n (%s)')
```

The variable φ evaluates to the type of ASTs of programs inhabiting the type corresponding to `T % repr(T)`, where `T` is `'(lambda T: □(T %% repr(T)))\n (%s)'`. What Martin–Löf type does this string, `T % repr(T)`, represent? It represents `□(T % repr(T))`, of course. Hence φ is the type of syntax trees of programs that produce proofs of φ—-in other words, φ is a Henkin sentence.

Taking it one step further, assume Python has a function `Π(a, b)` which takes two Python representations of Martin–Löf types and produces the Martin–Löf type `(a → b)` of functions from `a` to `b`. If we also assume that these functions exist in the term language of string representations of Martin–Löf types, we can consider the function

```
def Lob(X):
  T = '(lambda T: Π(□(T %% repr(T)), X))(%s)'
  φ = Π(□(T % repr(T)), X)
  return φ
```

What does `Lob(X)` return? It returns the type φ of abstract syntax trees of programs producing proofs that "if φ is provable, then `X`." Concretely, `Lob(⊥)` returns the type of programs which prove Martin–Löf type theory consistent, `Lob(SantaClaus)` returns the variant of the Santa Claus sentence that says "if this sentence is provable, then Santa Claus exists."

Let us now try producing the true Santa Claus sentence, the one that says "If this sentence is true, Santa Claus exists." We need a function `Eval` which takes a string representing a Martin–Löf program, and evaluates it to produce a term. Consider the Python program

```
def Tarski(X):
  T = '(lambda T: Π(Eval(T %% repr(T)), X)(%s)'
  φ = Π(Eval(T % repr(T)), X)
  return φ
```

Running `Eval(T % repr(T))` tries to produce a term that is the type of functions from `Eval(T % repr(T))` to `X`. Note that φ is itself the type of functions from `Eval(T % repr(T))` to `X`. If `Eval(T % repr(T))` could produce a term of type φ, then φ would evaluate to the type $\varphi \to$ `X`, giving us a bona fide Santa Claus sentence. However, `Eval(T % repr(T))` attempts to produce the type of functions from `Eval(T % repr(T))` to `X` by evaluating `Eval(T % repr(T))`. This throws the function `Tarski` into an infinite loop which never terminates. (Indeed, choosing `X` = ⊥ it's trivial to show that there's no way to write `Eval` such that `Tarski` halts, unless Martin–Löf type theory is inconsistent.)

3. Abstract Syntax Trees for Dependent Type Theory

The idea of formalizing a type of syntax trees which only permits well-typed programs is common in the literature (McBride 2010; Chapman 2009; Danielsson 2007). For example, here is a very simple (and incomplete) formalization with dependent function types (Π), a unit type ($\top$), an empty type ($\bot$), and functions (λ).

We will use some standard data type declarations, which are provided for completeness in Appendix A.

```
mutual
  infixl 2 _▷_

  data Context : Set where
    ε : Context
    _▷_ : (Γ : Context) → Type Γ → Context

  data Type : Context → Set where
    `⊤ : ∀ {Γ} → Type Γ
    `⊥ : ∀ {Γ} → Type Γ
    `Π : ∀ {Γ}
      → (A : Type Γ) → Type (Γ ▷ A) → Type Γ

  data Term : {Γ : Context} → Type Γ → Set where
    `tt : ∀ {Γ} → Term {Γ} `⊤
    `λ : ∀ {Γ A B} → Term B → Term {Γ} (`Π A B)
```

An easy way to check consistency of a syntactic theory which is weaker than the theory of the ambient proof assistant is to define an interpretation function, also commonly known as an unquoter, or a denotation function, from the syntax into the universe of types. This function gives a semantic model to the syntax. Here is an example of such a function:

```
mutual
  ⟦_⟧ᶜ : Context → Set
  ⟦ ε ⟧ᶜ = ⊤
  ⟦ Γ ▷ T ⟧ᶜ = Σ ⟦ Γ ⟧ᶜ ⟦ T ⟧ᵀ

  ⟦_⟧ᵀ : ∀ {Γ}
    → Type Γ → ⟦ Γ ⟧ᶜ → Set
  ⟦ `⊤ ⟧ᵀ Γ⇓ = ⊤
  ⟦ `⊥ ⟧ᵀ Γ⇓ = ⊥
  ⟦ `Π A B ⟧ᵀ Γ⇓ = (x : ⟦ A ⟧ᵀ Γ⇓) → ⟦ B ⟧ᵀ (Γ⇓ , x)

  ⟦_⟧ᵗ : ∀ {Γ T}
    → Term {Γ} T → (Γ⇓ : ⟦ Γ ⟧ᶜ) → ⟦ T ⟧ᵀ Γ⇓
  ⟦ `tt ⟧ᵗ Γ⇓ = tt
  ⟦ `λ f ⟧ᵗ Γ⇓ x = ⟦ f ⟧ᵗ (Γ⇓ , x)
```

Note that this interpretation function has an essential property that we will call *locality*: the interpretation of any given constructor does not require doing case analysis on any of its arguments. By contrast, one could imagine an interpretation function that interpreted function types differently depending on their domain and codomain; for example, one might interpret `(`⊥ `→ A)` as $\top$, or one might interpret an equality type differently at each type, as in Observational Type Theory (Altenkirch et al. 2007).

4. This Paper

In this paper, we make extensive use of this trick for validating models. In section 6, we formalize the simplest syntax that supports Löb's theorem and prove it sound relative to Agda in 12 lines of code; the understanding is that this syntax could be extended to support basically anything you might want. We then present in section 7 an extended version of this solution, which supports enough operations that we can prove our syntax sound (consistent), incomplete, and nonempty. In a hundred lines of code, we prove Löb's theorem in section 8 under the assumption that we are given a quine; this is basically the well-typed functional version of the program that uses `open(__file__, 'r').read()`. After taking a digression for an application of Löb's theorem to the prisoner's dilemma in section 9, we sketch in section 10 our implementation of Löb's theorem (code in the supplemental material) based on only the assumption that we can add a level of quotation to our syntax tree; this is the equivalent of letting the compiler implement `repr`, rather than implementing it ourselves. We close in section 11 with some discussion about avenues for removing the hard-coded `repr`.

5. Prior Work

There exist a number of implementations or formalizations of various flavors of Löb's theorem in the literature. Appel et al. use Löb's theorem as an induction rule for program logics in Coq (Appel et al. 2007). Piponi formalizes a rule with the same shape as Löb's theorem in Haskell, and uses it for, among other things, spreadsheet evaluation (Piponi 2006). Simmons and Toninho formalize a constructive provability logic in Agda, and prove Löb's theorem within that logic (Simmons and Toninho 2012).

Gödel's incompleteness theorems, easy corollaries to Löb's theorem, have been formally verified numerous times (Shankar 1986, 1997; O'Connor 2005; Paulson 2015).

To our knowledge, our twelve line proof is the shortest self-contained formally verified proof of the admissibility of Löb's theorem to date. We are not aware of other formally verified proofs of Löb's theorem which interpret the modal $\Box$ operator as an inductively defined type of syntax trees of proofs of a given theorem, as we do in this formalization, though presumably the modal $\Box$ operator Simmons and Toninho could be interpreted as such syntax trees. Finally, we are not aware of other work which uses the trick of talking about a local interpretation function (as described at the end of section 3) to talk about consistent extensions to classes of encodings of type theory.

6. Trivial Encoding

We begin with a language that supports almost nothing other than Löb's theorem. We use `□ T` to denote the type of `Terms` of whose syntactic type is `T`. We use `` `□ T `` to denote the syntactic type corresponding to the type of (syntactic) terms whose syntactic type is `T`. For example, the type of a `repr` which operated on syntax trees would be `□ T → □ (`□ T)`.

```
  data Type : Set where
    _`→`_ : Type → Type → Type
    `□ : Type → Type

  data □ : Type → Set where
    Löb : ∀ {X} → □ (`□ X `→ X) → □ X
```

The only term supported by our term language is Löb's theorem. We can prove this language consistent relative to Agda with an interpreter:

```
  ⟦_⟧ᵀ : Type → Set
  ⟦ A `→ B ⟧ᵀ = ⟦ A ⟧ᵀ → ⟦ B ⟧ᵀ
  ⟦ `□ T ⟧ᵀ = □ T

  ⟦_⟧ᵗ : ∀ {T : Type} → □ T → ⟦ T ⟧ᵀ
  ⟦ Löb □`X→X ⟧ᵗ = ⟦ □`X→X ⟧ᵗ (Löb □`X→X)
```

To interpret Löb's theorem applied to the syntax for a compiler f (`□`X→X` in the code above), we interpret $f$, and then apply this interpretation to the constructor `Löb` applied to f.

Finally, we tie it all together:

```
  löb : ∀ {`X} → □ (`□ `X `→ `X) → ⟦ `X ⟧ᵀ
  löb f = ⟦ Löb f ⟧ᵗ
```

This code is deceptively short, with all of the interesting work happening in the interpretation of `Löb`.

What have we actually proven, here? It may seem as though we've proven absolutely nothing, or it may seem as though we've proven that Löb's theorem always holds. Neither of these is the

case. The latter is ruled out, for example, by the existence of an self-interpreter for F_ω (Brown and Palsberg 2016).[2]

We have proven the following. Suppose you have a formalization of type theory which has a syntax for types, and a syntax for terms indexed over those types. If there is a "local explanation" for the system being sound, i.e., an interpretation function where each rule does not need to know about the full list of constructors, then it is consistent to add a constructor for Löb's theorem to your syntax. This means that it is impossible to contradict Löb's theorem no matter what (consistent) constructors you add. We will see in the next section how this gives incompleteness, and discuss in later sections how to *prove Löb's theorem*, rather than simply proving that it is consistent to assume.

7. Encoding with Soundness, Incompleteness, and Non-Emptiness

By augmenting our representation with top (`'⊤'`) and bottom (`'⊥'`) types, and a unique inhabitant of `'⊤'`, we can prove soundness, incompleteness, and non-emptiness.

```
data Type : Set where
  _'→'_ : Type → Type → Type
  '□' : Type → Type
  '⊤' : Type
  '⊥' : Type

-- "□" is sometimes written as "Term"
data □ : Type → Set where
  Löb : ∀ {X} → □ ('□' X '→' X) → □ X
  'tt' : □ '⊤'

⟦_⟧ᵀ : Type → Set
⟦ A '→' B ⟧ᵀ = ⟦ A ⟧ᵀ → ⟦ B ⟧ᵀ
⟦ '□' T ⟧ᵀ   = □ T
⟦ '⊤' ⟧ᵀ     = ⊤
⟦ '⊥' ⟧ᵀ     = ⊥

⟦_⟧ᵗ : ∀ {T : Type} → □ T → ⟦ T ⟧ᵀ
⟦ Löb □'X'→X ⟧ᵗ = ⟦ □'X'→X ⟧ᵗ (Löb □'X'→X)
⟦ 'tt' ⟧ᵗ = tt

'¬'_ : Type → Type
'¬' T = T '→' '⊥'

löb : ∀ {'X'} → □ ('□' 'X' '→' 'X') → ⟦ 'X' ⟧ᵀ
löb f = ⟦ Löb f ⟧ᵗ

-- There is no syntactic proof of absurdity
soundness : ¬ □ '⊥'
soundness x = ⟦ x ⟧ᵗ

-- but it would be absurd to have a syntactic
-- proof of that fact
incompleteness : ¬ □ ('¬' ('□' '⊥'))
incompleteness = löb
```

[2] One may wonder how exactly the self-interpreter for F_ω does not contradict this theorem. In private conversations with Matt Brown, we found that the F_ω self-interpreter does not have a separate syntax for types, instead indexing its terms over types in the metalanguage. This means that the type of Löb's theorem becomes either `□ (□ X → X) → □ X`, which is not strictly positive, or `□ (X → X) → □ X`, which, on interpretation, must be filled with a general fixpoint operator. Such an operator is well-known to be inconsistent.

```
-- However, there are syntactic proofs of some
-- things (namely ⊤)
non-emptiness : □ '⊤'
non-emptiness = 'tt'

-- There are no syntactic interpreters, things
-- which, at any type, evaluate code at that
-- type to produce its result.
no-interpreters : ¬ (∀ {'X'} → □ ('□' 'X' '→' 'X'))
no-interpreters interp = löb (interp {'⊥'})
```

What is this incompleteness theorem? Gödel's incompleteness theorem is typically interpreted as "there exist true but unprovable statements." In intuitionistic logic, this is hardly surprising. A more accurate rendition of the theorem in Agda might be "there exist true but inadmissible statements," i.e., there are statements which are provable meta-theoretically, but which lead to (meta-theoretically–provable) inconsistency if assumed at the object level.

This may seem a bit esoteric, so let's back up a bit, and make it more concrete. Let's begin by banishing "truth". Sometimes it is useful to formalize a notion of provability. For example, you might want to show neither assuming T nor assuming $\neg T$ yields a proof of contradiction. You cannot phrase this is $\neg T \wedge \neg\neg T$, for that is absurd. Instead, you want to say something like $(\neg\Box T)\wedge\neg\Box(\neg T)$, i.e., it would be absurd to have a proof object of either T or of $\neg T$. After a while, you might find that meta-programming in this formal syntax is nice, and you might want it to be able to formalize every proof, so that you can do all of your solving reflectively. If you're like us, you might even want to reason about the reflective tactics themselves in a reflective manner; you'd want to be able to add levels of quotation to quoted things to talk about such tactics.

The incompleteness theorem, then, is this: your reflective system, no matter how powerful, cannot formalize every proof. For any fixed language of syntactic proofs which is powerful enough to represent itself, there will always be some valid proofs that you cannot reflect into your syntax. In particular, you might be able to prove that your syntax has no proofs of $\bot$ (by interpreting any such proof). But you'll be unable to quote that proof. This is what the incompleteness theorem stated above says. Incompleteness, fundamentally, is a result about the limitations of formalizing provability.

8. Encoding with Quines

We now weaken our assumptions further. Rather than assuming Löb's theorem, we instead assume only a type-level quine in our representation. Recall that a *quine* is a program that outputs some function of its own source code. A *type-level quine at* ϕ is program that outputs the result of evaluating the function ϕ on the abstract syntax tree of its own type. Letting `Quine` ϕ denote the constructor for a type-level quine at ϕ, we have an isomorphism between `Quine` ϕ and ϕ `⌜ Quine` ϕ `⌝ᵀ`, where `⌜ Quine` ϕ `⌝ᵀ` is the abstract syntax tree for the source code of `Quine` ϕ. Note that we assume constructors for "adding a level of quotation", turning abstract syntax trees for programs of type T into abstract syntax trees for abstract syntax trees for programs of type T; this corresponds to `repr`.

We begin with an encoding of contexts and types, repeating from above the constructors of '→', '□', '⊤', and '⊥'. We add to this a constructor for quines (`Quine`), and a constructor for syntax trees of types in the empty context ('Typeε'). Finally, rather than proving weakening and substitution as mutually recursive definitions, we take the easier but more verbose route of adding constructors that allow adding and substituting extra terms in the context. Note that '□' is now a function of the represented language, rather than a meta-level operator.

Note that we use the infix operator `_‘’_` to denote substitution.

```
mutual
  data Context : Set where
    ε : Context
    _▷_ : (Γ : Context) → Type Γ → Context

  data Type : Context → Set where
    _‘→’_ : ∀ {Γ} → Type Γ → Type Γ → Type Γ
    ‘⊤’ : ∀ {Γ} → Type Γ
    ‘⊥’ : ∀ {Γ} → Type Γ
    ‘Type’ : ∀ {Γ} → Type Γ
    ‘□’ : ∀ {Γ} → Type (Γ ▷ ‘Type’)
    Quine : Type (ε ▷ ‘Type’) → Type ε
    W : ∀ {Γ A}
      → Type Γ → Type (Γ ▷ A)
    W₁ : ∀ {Γ A B}
      → Type (Γ ▷ B) → Type (Γ ▷ A ▷ (W B))
    _‘’_ : ∀ {Γ A}
      → Type (Γ ▷ A) → Term A → Type Γ
```

In addition to ‘λ’ and ‘tt’, we now have the AST-equivalents of Python’s `repr`, which we denote as $\ulcorner \_ \urcorner^{\mathsf{T}}$ for the type-level add-quote function, and $\ulcorner \_ \urcorner^{\mathsf{t}}$ for the term-level add-quote function. We add constructors `quine→` and `quine←` that exhibit the isomorphism that defines our type-level quine constructor, though we elide a constructor declaring that these are inverses, as we found it unnecessary.

To construct the proof of Löb’s theorem, we need a few other standard constructors, such as ‘$\mathtt{VAR}_0$’, which references a term in the context; `_‘’ₐ_`, which we use to denote function application; `_‘∘’_`, a function composition operator; and ‘$\ulcorner$‘$\mathtt{VAR}_0$’$\urcorner^{\mathsf{t}}$’, the variant of ‘$\mathtt{VAR}_0$’ which adds an extra level of syntax-trees. We also include a number of constructors that handle weakening and substitution; this allows us to avoid both inductive-recursive definitions of weakening and substitution, and avoid defining a judgmental equality or conversion relation.

```
  data Term : {Γ : Context} → Type Γ → Set where
    ‘λ’ : ∀ {Γ A B}
      → Term {Γ ▷ A} (W B) → Term (A ‘→’ B)
    ‘tt’ : ∀ {Γ}
      → Term {Γ} ‘⊤’
    ⌜_⌝ᵀ : ∀ {Γ} -- type-level repr
      → Type ε
      → Term {Γ} ‘Type’
    ⌜_⌝ᵗ : ∀ {Γ T} -- term-level repr
      → Term {ε} T
      → Term {Γ} (‘□’ ‘’ ⌜ T ⌝ᵀ)
    quine→ : ∀ {ϕ}
      → Term {ε} (Quine ϕ ‘→’ ϕ ‘’ ⌜ Quine ϕ ⌝ᵀ)
    quine← : ∀ {ϕ}
      → Term {ε} (ϕ ‘’ ⌜ Quine ϕ ⌝ᵀ ‘→’ Quine ϕ)
    -- The constructors below here are for
    -- variables, weakening, and substitution
    ‘VAR₀’ : ∀ {Γ T}
      → Term {Γ ▷ T} (W T)
    _‘’ₐ_ : ∀ {Γ A B}
      → Term {Γ} (A ‘→’ B)
      → Term {Γ} A
      → Term {Γ} B
    _‘∘’_ : ∀ {Γ A B C}
      → Term {Γ} (B ‘→’ C)
      → Term {Γ} (A ‘→’ B)
      → Term {Γ} (A ‘→’ C)
    ‘⌜‘VAR₀’⌝ᵗ’ : ∀ {T}
      → Term {ε ▷ ‘□’ ‘’ ⌜ T ⌝ᵀ}
        (W (‘□’ ‘’ ⌜ ‘□’ ‘’ ⌜ T ⌝ᵀ ⌝ᵀ))
    →SW₁SV→W : ∀ {Γ T X A B} {x : Term {Γ} X}
      → Term (T ‘→’ (W₁ A ‘’ ‘VAR₀’ ‘→’ W B) ‘’ x)
      → Term (T ‘→’ A ‘’ x ‘→’ B)
    ←SW₁SV→W : ∀ {Γ T X A B} {x : Term {Γ} X}
      → Term ((W₁ A ‘’ ‘VAR₀’ ‘→’ W B) ‘’ x ‘→’ T)
      → Term ((A ‘’ x ‘→’ B) ‘→’ T)
    w : ∀ {Γ A T} → Term A → Term {Γ ▷ T} (W A)
    w→ : ∀ {Γ A B X}
      → Term {Γ} (A ‘→’ B)
      → Term {Γ ▷ X} (W A ‘→’ W B)
    _w‘’’’ₐ_ : ∀ {A B T}
      → Term {ε ▷ T} (W (‘□’ ‘’ ⌜ A ‘→’ B ⌝ᵀ))
      → Term {ε ▷ T} (W (‘□’ ‘’ ⌜ A ⌝ᵀ))
      → Term {ε ▷ T} (W (‘□’ ‘’ ⌜ B ⌝ᵀ))

  □ : Type ε → Set _
  □ = Term {ε}
```

To verify the soundness of our syntax, we provide a model for it and an interpretation into that model. We call particular attention to the interpretation of ‘□’, which is just `Term {ε}`; to `Quine ϕ`, which is the interpretation of ϕ applied to `Quine ϕ`; and to the interpretations of the quine isomorphism functions, which are just the identity functions.

```
  max-level : Level
  max-level = lzero -- also works for higher levels

  mutual
    ⟦_⟧ᶜ : (Γ : Context) → Set (lsuc max-level)
    ⟦ ε ⟧ᶜ = ⊤
    ⟦ Γ ▷ T ⟧ᶜ = Σ ⟦ Γ ⟧ᶜ ⟦ T ⟧ᵀ

    ⟦_⟧ᵀ : ∀ {Γ}
      → Type Γ → ⟦ Γ ⟧ᶜ → Set max-level
    ⟦ ‘Type’ ⟧ᵀ Γ⇓ = Lifted (Type ε)
    ⟦ ‘□’ ⟧ᵀ Γ⇓ = Lifted (Term {ε} (lower (snd Γ⇓)))
    ⟦ Quine ϕ ⟧ᵀ Γ⇓ = ⟦ ϕ ⟧ᵀ (Γ⇓ , lift (Quine ϕ))
    -- The rest of the type-level interpretations
    -- are the obvious ones, if a bit obscured by
    -- carrying around the context.
    ⟦ A ‘→’ B ⟧ᵀ Γ⇓ = ⟦ A ⟧ᵀ Γ⇓ → ⟦ B ⟧ᵀ Γ⇓
    ⟦ ‘⊤’ ⟧ᵀ Γ⇓ = ⊤
    ⟦ ‘⊥’ ⟧ᵀ Γ⇓ = ⊥
    ⟦ W T ⟧ᵀ Γ⇓ = ⟦ T ⟧ᵀ (fst Γ⇓)
    ⟦ W₁ T ⟧ᵀ Γ⇓ = ⟦ T ⟧ᵀ (fst (fst Γ⇓) , snd Γ⇓)
    ⟦ T ‘’ x ⟧ᵀ Γ⇓ = ⟦ T ⟧ᵀ (Γ⇓ , ⟦ x ⟧ᵗ Γ⇓)

    ⟦_⟧ᵗ : ∀ {Γ T}
      → Term {Γ} T → (Γ⇓ : ⟦ Γ ⟧ᶜ) → ⟦ T ⟧ᵀ Γ⇓
    ⟦ ⌜ x ⌝ᵀ ⟧ᵗ Γ⇓ = lift x
    ⟦ ⌜ x ⌝ᵗ ⟧ᵗ Γ⇓ = lift x
    ⟦ quine→ ⟧ᵗ Γ⇓ x = x
    ⟦ quine← ⟧ᵗ Γ⇓ x = x
    -- The rest of the term-level interpretations
    -- are the obvious ones, if a bit obscured by
    -- carrying around the context.
    ⟦ ‘λ’ f ⟧ᵗ Γ⇓ x = ⟦ f ⟧ᵗ (Γ⇓ , x)
    ⟦ ‘tt’ ⟧ᵗ Γ⇓ = tt
```

```
⟦ 'VAR₀' ⟧ᵗ Γ⇓ = snd Γ⇓
⟦ '⌜'VAR₀'⌝ᵗ' ⟧ᵗ Γ⇓ = lift ⌜ lower (snd Γ⇓) ⌝ᵗ
⟦ g '∘' f ⟧ᵗ Γ⇓ x = ⟦ g ⟧ᵗ Γ⇓ (⟦ f ⟧ᵗ Γ⇓ x)
⟦ f ''ₐ x ⟧ᵗ Γ⇓ = ⟦ f ⟧ᵗ Γ⇓ (⟦ x ⟧ᵗ Γ⇓)
⟦ ←SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
⟦ →SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
⟦ w x ⟧ᵗ Γ⇓ = ⟦ x ⟧ᵗ (fst Γ⇓)
⟦ w→ f ⟧ᵗ Γ⇓ = ⟦ f ⟧ᵗ (fst Γ⇓)
⟦ f w''''ₐ x ⟧ᵗ Γ⇓
  = lift (lower (⟦ f ⟧ᵗ Γ⇓) ''ₐ lower (⟦ x ⟧ᵗ Γ⇓))
```

To prove Löb's theorem, we must create the sentence "if this sentence is provable, then X", and then provide and inhabitant of that type. We can define this sentence, which we call `'H'`, as the type-level quine at the function $\lambda v.\ \Box v \to$ 'X'. We can then convert back and forth between the types $\Box$ `'H'` and $\Box$ `'H'` $\to$ `'X'` with our quine isomorphism functions, and a bit of quotation magic and function application gives us a term of type $\Box$ `'H'` $\to$ $\Box$ `'X'`; this corresponds to the inference of the provability of Santa Claus' existence from the assumption that the sentence is provable. We compose this with the assumption of Löb's theorem, that $\Box$ `'X'` $\to$ `'X'`, to get a term of type $\Box$ `'H'` $\to$ `'X'`, i.e., a term of type `'H'`; this is the inference that when provability implies truth, Santa Claus exists, and hence that the sentence is provable. Finally, we apply this to its own quotation, obtaining a term of type $\Box$ `'X'`, i.e., a proof that Santa Claus exists.

```
module inner ('X' : Type ε)
       ('f' : Term {ε} ('□' '' ⌜ 'X' ⌝ᵀ '→' 'X'))
  where
  'H' : Type ε
  'H' = Quine (W₁ '□' '' 'VAR₀' '→' W 'X')

  'toH' : □ (('□' '' ⌜ 'H' ⌝ᵀ '→' 'X') '→' 'H')
  'toH' = ←SW₁SV→W quine←

  'fromH' : □ ('H' '→' ('□' '' ⌜ 'H' ⌝ᵀ '→' 'X'))
  'fromH' = →SW₁SV→W quine→

  '□'H'→□'X'' : □ ('□' '' ⌜ 'H' ⌝ᵀ '→' '□' '' ⌜ 'X' ⌝ᵀ)
  '□'H'→□'X''
    = 'λ' (w ⌜ 'fromH' ⌝ᵗ
      w''''ₐ 'VAR₀'
      w''''ₐ '⌜'VAR₀'⌝ᵗ')

  'h' : Term 'H'
  'h' = 'toH' ''ₐ ('f' '∘' '□'H'→□'X'')

  Löb : □ 'X'
  Löb = 'fromH' ''ₐ 'h' ''ₐ ⌜ 'h' ⌝ᵗ

Löb : ∀ {X} → □ ('□' '' ⌜ X ⌝ᵀ '→' X) → □ X
Löb {X} f = inner.Löb X f

⟦_⟧ : Type ε → Set _
⟦ T ⟧ = ⟦ T ⟧ᵀ tt

'¬'_ : ∀ {Γ} → Type Γ → Type Γ
'¬' T = T '→' '⊥'

löb : ∀ {'X'} → □ ('□' '' ⌜ 'X' ⌝ᵀ '→' 'X') → ⟦ 'X' ⟧
löb f = ⟦_⟧ᵗ (Löb f) tt
```

B Says \ A Says	Cooperate	Defect
Cooperate	(1 year, 1 year)	(0 years, 3 years)
Defect	(3 years, 0 years)	(2 years, 2 years)

Table 2. The payoff matrix for the prisoner's dilemma; each cell contains (the years A spends in prison, the years B spends in prison).

As above, we can again prove soundness, incompleteness, and non-emptiness.

```
incompleteness : ¬ □ ('¬' ('□' '' ⌜ '⊥' ⌝ᵀ))
incompleteness = löb

soundness : ¬ □ '⊥'
soundness x = ⟦ x ⟧ᵗ tt

non-emptiness : Σ (Type ε) (λ T → □ T)
non-emptiness = '⊤' , 'tt'
```

9. Digression: Application of Quining to The Prisoner's Dilemma

In this section, we use a slightly more enriched encoding of syntax; see Appendix B for details.

9.1 The Prisoner's Dilemma

The Prisoner's Dilemma is a classic problem in game theory. Two people have been arrested as suspects in a crime and are being held in solitary confinement, with no means of communication. The investigators offer each of them a plea bargain: a decreased sentence for ratting out the other person. Each suspect can then choose to either cooperate with the other suspect by remaining silent, or defect by ratting out the other suspect. The possible outcomes are summarized in Table 2.

Suspect A might reason thusly: "Suppose the other suspect cooperates with me. Then I'd get off with no prison time if I defected, while I'd have to spend a year in prison if I cooperate. Similarly, if the other suspect defects, then I'd get two years in prison for defecting, and three for cooperating. In all cases, I do better by defecting." If suspect B reasons similarly, then both decide to defect, and both get two years in prison, despite the fact that both prefer the (Cooperate, Cooperate) outcome over the (Defect, Defect) outcome!

9.2 Adding Source Code

We have the intuition that if both suspects are good at reasoning, and both know that they'll reason the same way, then they should be able to mutually cooperate. One way to formalize this is to talk about programs (rather than people) playing the prisoner's dilemma, and to allow each program access to its own source code and its opponent's source code (Barasz et al. 2014).

We have formalized this framework in Agda: we use `'Bot'` to denote the type of programs that can play in such a prisoner's dilemma; each one takes in source code for two `'Bot'`s and outputs a proposition which is true (a type which is inhabited) if and only if it cooperates with its opponent. Said another way, the output of each bot is a proposition describing the assertion that it cooperates with its opponent.

```
open lob

-- 'Bot' is defined as the fixed point of
```

```
-- 'Bot'
-- ↔ (Term 'Bot' → Term 'Bot' → 'Type')
'Bot' : ∀ {Γ} → Type Γ
'Bot' {Γ}
  = Quine (W₁ 'Term' '' 'VAR₀'
      '→' W₁ 'Term' '' 'VAR₀'
      '→' W ('Type' Γ))
```

To construct an executable bot, we could do a bounded search for proofs of this proposition; one useful method described in (Barasz et al. 2014) is to use Kripke frames. This computation is, however, beyond the scope of this paper.

The assertion that a bot $\mathtt{b}_1$ cooperates with a bot $\mathtt{b}_2$ is the result of interpreting the source code for the bot, and feeding the resulting function the source code for $\mathtt{b}_1$ and $\mathtt{b}_2$.

```
-- N.B. "□" means "Term {ε}", i.e., a term in
-- the empty context
_cooperates-with_ : □ 'Bot' → □ 'Bot' → Type ε
b₁ cooperates-with b₂ = lower (⟦ b₁ ⟧ᵗ tt (lift b₁) (lift b₂))
```

We now provide a convenience constructor for building bots, based on the definition of quines, and present three relatively simple bots: DefectBot, CooperateBot, and FairBot.

```
make-bot : ∀ {Γ}
  → Term {Γ ▷ '□' 'Bot' ▷ W ('□' 'Bot')}
      (W (W ('Type' Γ)))
  → Term {Γ} 'Bot'
make-bot t
  = ←SW₁SV→SW₁SV→W
    quine← '' ' 'λ' (→w ('λ' t))

'DefectBot'  : □ 'Bot'
'CooperateBot' : □ 'Bot'
'FairBot'    : □ 'Bot'
```

The first two bots are very simple: DefectBot never cooperates (the assertion that DefectBot cooperates is a contradiction), while CooperateBot always cooperates. We define these bots, and prove that DefectBot never cooperates and CooperateBot always cooperates.

```
'DefectBot'  = make-bot (w (w ⌜ '⊥' ⌝ᵀ))
'CooperateBot' = make-bot (w (w ⌜ '⊤' ⌝ᵀ))

DB-defects : ∀ {b}
  → ¬ ⟦ 'DefectBot' cooperates-with b ⟧
DB-defects {b} pf = pf

CB-cooperates : ∀ {b}
  → ⟦ 'CooperateBot' cooperates-with b ⟧
CB-cooperates {b} = tt
```

We can do better than DefectBot, though, now that we have source code. FairBot cooperates with you if and only if it can find a proof that you cooperate with FairBot. By Löb's theorem, to prove that FairBot cooperates with itself, it suffices to prove that if there is a proof that FairBot cooperates with itself, then FairBot does, in fact, cooperate with itself. This is obvious, though: FairBot decides whether or not to cooperate with itself by searching for a proof that it does, in fact, cooperate with itself.

To define FairBot, we first define what it means for the other bot to cooperate with some particular bot.

```
-- We can "evaluate" a bot to turn it into a
-- function accepting the source code of two
-- bots.
'eval-bot' : ∀ {Γ}
  → Term {Γ} ('Bot'
      '→' ('□' 'Bot' '→' '□' 'Bot' '→' 'Type' Γ))
'eval-bot' = →SW₁SV→SW₁SV→W quine→

-- We can quote this, and get a function that
-- takes the source code for a bot, and
-- outputs the source code for a function that
-- takes (the source code for) that bot's
-- opponent, and returns an assertion of
-- cooperation with that opponent
''eval-bot'' : ∀ {Γ}
  → Term {Γ} ('□' 'Bot'
      '→' '□' ({- other -} '□' 'Bot' '→' 'Type' Γ))
''eval-bot''
  = 'λ' (w ⌜ 'eval-bot' ⌝ᵗ
      w'''' 'VAR₀'
      w'''' ⌜'VAR₀'⌝ᵗ')

-- The assertion "our opponent cooperates with
-- a bot b" is equivalent to the evaluation of
-- our opponent, applied to b. Most of the
-- noise in this statement is manipulation of
-- weakening and substitution.
'other-cooperates-with' : ∀ {Γ}
  → Term {Γ
      ▷ '□' 'Bot'
      ▷ W ('□' 'Bot')}
      (W (W ('□' 'Bot')) '→' W (W ('□' ('Type' Γ))))
'other-cooperates-with' {Γ}
  = 'eval-other''
      '∘' w→ (w (w→ (w ('λ' ⌜'VAR₀'⌝ᵗ'))))
  where
    'eval-other'
      : Term {Γ ▷ '□' 'Bot' ▷ W ('□' 'Bot')}
          (W (W ('□' ('□' 'Bot' '→' 'Type' Γ))))
    'eval-other'
      = w→ (w (w→ (w ''eval-bot''))) '' 'VAR₀'

    'eval-other''
      : Term (W (W ('□' ('□' 'Bot'))))
          '→' W (W ('□' ('Type' Γ))))
    'eval-other''
      = ww→ (w→ (w (w→ (w '''')))) '' 'eval-other')

-- A bot gets its own source code as the first
-- argument (of two)
'self' : ∀ {Γ}
  → Term {Γ ▷ '□' 'Bot' ▷ W ('□' 'Bot')}
      (W (W ('□' 'Bot')))
'self' = w 'VAR₀'

-- A bot gets its opponent's source code as
-- the second argument (of two)
'other' : ∀ {Γ}
  → Term {Γ ▷ '□' 'Bot' ▷ W ('□' 'Bot')}
      (W (W ('□' 'Bot')))
'other' = 'VAR₀'

-- FairBot is the bot that cooperates iff its
-- opponent cooperates with it
'FairBot'
  = make-bot (''□'' ('other-cooperates-with' '' 'self'))
```

We leave the proof that this formalization of FairBot cooperates with itself as an exercise for the reader. In Appendix C, we present

an alternative formalization with a simple proof that FairBot cooperates with itself, but with no general definition of the type of bots; we relegate this code to an appendix so as to not confuse the reader by introducing a different way of handling contexts and weakening in the middle of this paper.

10. Encoding with an Add-Quote Function

Now we return to our proving of Löb's theorem. Included in the artifact for this paper[3] is code that replaces the `Quine` constructor with simpler constructors. Because the lack of β-reduction in the syntax clouds the main points and makes the code rather verbose, we do not include the code in the paper, and instead describe the most interesting and central points.

Recall our Python quine from Table 2:

```
(lambda T: Π(□(T % repr(T)), X))
 ('(lambda T: Π(□(T %% repr(T)), X))\n (%s)')
```

To translate this into Agda, we need to give a type to `T`. Clearly, `T` needs to be of type `Type ???` for some context `???`. Since we need to be able to substitute something into that context, we must have `T : Type (Γ ▷ ???)`, i.e., `T` must be a syntax tree for a type, with a hole in it.

What's the shape of the thing being substituted? Well, it's a syntax tree for a type with a hole in it. What shape does that hole have? The shape is that of a syntax tree with a hole in it... Uh-oh. Our quine's type, naïvely, is infinite!

We know of two ways to work around this. Classical mathematics, which uses Gödel codes instead of abstract syntax trees, uses an untyped representation of proofs. It's only later in the proof of Löb's theorem that a notion of a formula being "well-formed" is introduced.

Here, we describe an alternate approach. Rather than giving up types all-together, we can "box" the type of the hole, to hide it. Using `fst` and `snd` to denote projections from a Σ type, using `⌜ A ⌝` to denote the abstract syntax tree for `A`,[4] and using `%s` to denote the first variable in the context (written as '$\mathtt{VAR}_0$' in previous formalizations above), we can write:

```
dummy : Type (ε ▷ ⌜Σ Context Type⌝)
repr : Σ Context Type → Term {ε} ⌜Σ Context Type⌝

cast-fst
  : Σ Context Type → Type (ε ▷ ⌜Σ Context Type⌝)
cast-fst (ε ▷ ⌜Σ Context Type⌝ , T) = T
cast-fst (_ , _) = dummy

LöbSentence : Type ε
LöbSentence
  = (λ (T : Σ Context Type)
       → □ (cast-fst T % repr T) '→' X)
     ( ε ▷ ⌜Σ Context Type⌝
     , ⌜ (λ (T : Σ Context Type)
            → □ (cast-fst T % repr T) '→' X)
          (%s) ⌝
```

In this pseudo-Agda code, `cast-fst` unboxes the sentence that it gets, and returns it if it is the right type. Since the sentence is, in fact, always the right type, what we do in the other cases doesn't matter.

Summing up, the key ingredients to this construction are:

- A type of syntactic terms indexed over a type of syntactic types (and contexts)
- Decidable equality on syntactic contexts at a particular point (in particular, at Σ `Context Type`), with appropriate reduction on equal things
- Σ types, projections, and appropriate reduction on their projections
- Function types
- A function `repr` which adds a level of quotation to any syntax tree
- Syntax trees for all of the above

In any formalization of dependent type theory with all of these ingredients, we can prove Löb's theorem.

11. Conclusion

What remains to be done is formalizing Martin–Löf type theory without assuming `repr` and without assuming a constructor for the type of syntax trees (`'Context'`, `'Type'`, and `'Term'` or `'□'` in our formalizations). We would instead support inductive types, and construct these operators as inductive types and as folds over inductive types.

If you take away only three things from this paper, take away these:

1. There will always be some true things which are not possible to say, no matter how good you are at talking in type theory about type theory.
2. Giving meaning to syntax in a way that doesn't use cases inside cases allows you to talk about when it's okay to add new syntax.
3. If believing in something is enough to make it true, then it already is. Dream big.

A. Standard Data-Type Declarations

```
open import Agda.Primitive public
  using (Level; _⊔_; lzero; lsuc)

infixl 1 _,_
infixr 2 _×_
infixl 1 _≡_

record ⊤ {ℓ} : Set ℓ where
  constructor tt

data ⊥ {ℓ} : Set ℓ where

¬_ : ∀ {ℓ ℓ'} → Set ℓ → Set (ℓ ⊔ ℓ')
¬_ {ℓ} {ℓ'} T = T → ⊥ {ℓ'}

record Σ {a p} (A : Set a) (P : A → Set p)
       : Set (a ⊔ p)
  where
  constructor _,_
  field
    fst : A
    snd : P fst

open Σ public

data Lifted {a b} (A : Set a) : Set (b ⊔ a) where
  lift : A → Lifted A
```

[3] In `lob-build-quine.lagda`.

[4] Note that `⌜_⌝` would not be a function in the language, but a meta-level operation.

```
lower : ∀ {a b A} → Lifted {a} {b} A → A
lower (lift x) = x

_×_ : ∀ {ℓ ℓ′} (A : Set ℓ) (B : Set ℓ′) → Set (ℓ ⊔ ℓ′)
A × B = Σ A (λ _ → B)

data _≡_ {ℓ} {A : Set ℓ} (x : A) : A → Set ℓ where
  refl : x ≡ x

sym : {A : Set} → {x : A} → {y : A} → x ≡ y → y ≡ x
sym refl = refl

trans : {A : Set} → {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl

transport : ∀ {A : Set} {x : A} {y : A} → (P : A → Set)
  → x ≡ y → P x → P y
transport P refl v = v

data List (A : Set) : Set where
  ε : List A
  _::_ : A → List A → List A

_++_ : ∀{A} → List A → List A → List A
ε ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

B. Encoding of Löb's Theorem for the Prisoner's Dilemma

```
module lob where
  infixl 2 _▷_
  infixl 3 _‘’_
  infixr 1 _‘→’_
  infixr 1 _‘‘→’’_
  infixr 1 _‘‘‘→’’’_
  infixl 3 _‘’ₐ_
  infixl 3 _w‘‘’’ₐ_
  infixr 2 _‘∘’_
  infixr 2 _‘×’_
  infixr 2 _‘‘×’’_
  infixr 2 _w‘‘×’’_

  mutual
    data Context : Set where
      ε : Context
      _▷_ : (Γ : Context) → Type Γ → Context

    data Type : Context → Set where
      ‘⊤’ : ∀ {Γ} → Type Γ
      ‘⊥’ : ∀ {Γ} → Type Γ
      _‘→’_ : ∀ {Γ} → Type Γ → Type Γ → Type Γ
      _‘×’_ : ∀ {Γ} → Type Γ → Type Γ → Type Γ
      ‘Type’ : ∀ Γ → Type Γ
      ‘Term’ : ∀ {Γ} → Type (Γ ▷ ‘Type’ Γ)
      Quine : ∀ {Γ} → Type (Γ ▷ ‘Type’ Γ) → Type Γ
      W : ∀ {Γ A} → Type Γ → Type (Γ ▷ A)
      W₁ : ∀ {Γ A B}
        → Type (Γ ▷ B)
        → Type (Γ ▷ A ▷ W B)
      _‘’_ : ∀ {Γ A}
        → Type (Γ ▷ A)
        → Term A
        → Type Γ

  data Term : {Γ : Context} → Type Γ → Set where
    ‘tt’ : ∀ {Γ} → Term {Γ} ‘⊤’
    ‘λ’ : ∀ {Γ A B}
      → Term {Γ ▷ A} (W B)
      → Term (A ‘→’ B)
    ‘VAR₀’ : ∀ {Γ T} → Term {Γ ▷ T} (W T)
    ⌜_⌝ᵀ : ∀ {Γ}
      → Type Γ
      → Term {Γ} (‘Type’ Γ)
    ⌜_⌝ᵗ : ∀ {Γ T}
      → Term {Γ} T
      → Term {Γ} (‘Term’ ‘’ ⌜ T ⌝ᵀ)
    ‘⌜‘VAR₀’⌝ᵗ’ : ∀ {Γ T}
      → Term {Γ ▷ ‘Term’ ‘’ ⌜ T ⌝ᵀ}
        (W (‘Term’ ‘’ ⌜ ‘Term’ ‘’ ⌜ T ⌝ᵀ ⌝ᵀ))
    ‘⌜‘VAR₀’⌝ᵀ’ : ∀ {Γ}
      → Term {Γ ▷ ‘Type’ Γ}
        (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ))
    _‘’ₐ_ : ∀ {Γ A B}
      → Term {Γ} (A ‘→’ B)
      → Term {Γ} A
      → Term {Γ} B
    ‘‘×’’’ : ∀ {Γ}
      → Term {Γ} (‘Type’ Γ
        ‘→’ ‘Type’ Γ
        ‘→’ ‘Type’ Γ)
    quine→ : ∀ {Γ φ}
      → Term {Γ}
        (Quine φ ‘→’ φ ‘’ ⌜ Quine φ ⌝ᵀ)
    quine← : ∀ {Γ φ}
      → Term {Γ}
        (φ ‘’ ⌜ Quine φ ⌝ᵀ ‘→’ Quine φ)
    SW : ∀ {Γ X A} {a : Term A}
      → Term {Γ} (W X ‘’ a)
      → Term X
    →SW₁SV→W
      : ∀ {Γ T X A B} {x : Term X}
      → Term {Γ}
        (T ‘→’ (W₁ A ‘’ ‘VAR₀’ ‘→’ W B) ‘’ x)
      → Term {Γ}
        (T ‘→’ A ‘’ x ‘→’ B)
    ←SW₁SV→W
      : ∀ {Γ T X A B} {x : Term X}
      → Term {Γ}
        ((W₁ A ‘’ ‘VAR₀’ ‘→’ W B) ‘’ x ‘→’ T)
      → Term {Γ}
        ((A ‘’ x ‘→’ B) ‘→’ T)
    →SW₁SV→SW₁SV→W
      : ∀ {Γ T X A B} {x : Term X}
      → Term {Γ} (T ‘→’ (W₁ A ‘’ ‘VAR₀’
        ‘→’ W₁ A ‘’ ‘VAR₀’
        ‘→’ W B) ‘’ x)
      → Term {Γ} (T ‘→’ A ‘’ x ‘→’ A ‘’ x ‘→’ B)
    ←SW₁SV→SW₁SV→W
      : ∀ {Γ T X A B} {x : Term X}
      → Term {Γ} ((W₁ A ‘’ ‘VAR₀’
```

```
        ‘→’ W₁ A ‘’ ‘VAR₀’
        ‘→’ W B) ‘’ x
        ‘→’ T)
    → Term {Γ} ((A ‘’ x ‘→’ A ‘’ x ‘→’ B) ‘→’ T)
w : ∀ {Γ A T}
  → Term {Γ} A
  → Term {Γ ▷ T} (W A)
w→ : ∀ {Γ A B X}
  → Term {Γ ▷ X} (W (A ‘→’ B))
  → Term {Γ ▷ X} (W A ‘→’ W B)
→w : ∀ {Γ A B X}
  → Term {Γ ▷ X} (W A ‘→’ W B)
  → Term {Γ ▷ X} (W (A ‘→’ B))
ww→ : ∀ {Γ A B X Y}
  → Term {Γ ▷ X ▷ Y} (W (W (A ‘→’ B)))
  → Term {Γ ▷ X ▷ Y} (W (W A) ‘→’ W (W B))
→ww : ∀ {Γ A B X Y}
  → Term {Γ ▷ X ▷ Y} (W (W A) ‘→’ W (W B))
  → Term {Γ ▷ X ▷ Y} (W (W (A ‘→’ B)))
_‘∘’_ : ∀ {Γ A B C}
  → Term {Γ} (B ‘→’ C)
  → Term {Γ} (A ‘→’ B)
  → Term {Γ} (A ‘→’ C)
_w‘’’ₐ_ : ∀ {Γ A B T}
  → Term {Γ ▷ T} (W (‘Term’ ‘’ ⌜ A ‘→’ B ⌝ᵀ))
  → Term {Γ ▷ T} (W (‘Term’ ‘’ ⌜ A ⌝ᵀ))
  → Term {Γ ▷ T} (W (‘Term’ ‘’ ⌜ B ⌝ᵀ))
‘’’ₐ’ : ∀ {Γ A B}
  → Term {Γ} (‘Term’ ‘’ ⌜ A ‘→’ B ⌝ᵀ
    ‘→’ ‘Term’ ‘’ ⌜ A ⌝ᵀ
    ‘→’ ‘Term’ ‘’ ⌜ B ⌝ᵀ)
‘‘□’’ : ∀ {Γ A B}
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ)))
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Type’ Γ)))
_‘‘→’’_ : ∀ {Γ}
  → Term {Γ} (‘Type’ Γ)
  → Term {Γ} (‘Type’ Γ)
  → Term {Γ} (‘Type’ Γ)
_‘‘‘→’’’_ : ∀ {Γ A B}
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ)))
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ)))
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ)))
_‘‘‘×’’’_ : ∀ {Γ A B}
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ)))
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ)))
  → Term {Γ ▷ A ▷ B}
    (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ ⌝ᵀ)))

□ : Type ε → Set _
□ = Term {ε}

‘□’ : ∀ {Γ} → Type Γ → Type Γ
‘□’ T = ‘Term’ ‘’ ⌜ T ⌝ᵀ
```

```
_‘‘×’’_ : ∀ {Γ}
  → Term {Γ} (‘Type’ Γ)
  → Term {Γ} (‘Type’ Γ)
  → Term {Γ} (‘Type’ Γ)
A ‘‘×’’ B = ‘‘×’’’ ‘’ₐ A ‘’ₐ B

max-level : Level
max-level = lzero

mutual
  ⟦_⟧ᶜ : (Γ : Context) → Set (lsuc max-level)
  ⟦ ε ⟧ᶜ = ⊤
  ⟦ Γ ▷ T ⟧ᶜ = Σ ⟦ Γ ⟧ᶜ ⟦ T ⟧ᵀ

  ⟦_⟧ᵀ : {Γ : Context}
    → Type Γ
    → ⟦ Γ ⟧ᶜ
    → Set max-level
  ⟦ W T ⟧ᵀ ⟦Γ⟧
    = ⟦ T ⟧ᵀ (Σ.fst ⟦Γ⟧)
  ⟦ W₁ T ⟧ᵀ ⟦Γ⟧
    = ⟦ T ⟧ᵀ (Σ.fst (Σ.fst ⟦Γ⟧) , Σ.snd ⟦Γ⟧)
  ⟦ T ‘’ x ⟧ᵀ ⟦Γ⟧ = ⟦ T ⟧ᵀ (⟦Γ⟧ , ⟦ x ⟧ᵗ ⟦Γ⟧)
  ⟦ ‘Type’ Γ ⟧ᵀ ⟦Γ⟧
    = Lifted (Type Γ)
  ⟦ ‘Term’ ⟧ᵀ ⟦Γ⟧
    = Lifted (Term (lower (Σ.snd ⟦Γ⟧)))
  ⟦ A ‘→’ B ⟧ᵀ ⟦Γ⟧ = ⟦ A ⟧ᵀ ⟦Γ⟧ → ⟦ B ⟧ᵀ ⟦Γ⟧
  ⟦ A ‘×’ B ⟧ᵀ ⟦Γ⟧ = ⟦ A ⟧ᵀ ⟦Γ⟧ × ⟦ B ⟧ᵀ ⟦Γ⟧
  ⟦ ‘⊤’ ⟧ᵀ ⟦Γ⟧ = ⊤
  ⟦ ‘⊥’ ⟧ᵀ ⟦Γ⟧ = ⊥
  ⟦ Quine φ ⟧ᵀ ⟦Γ⟧ = ⟦ φ ⟧ᵀ (⟦Γ⟧ , (lift (Quine φ)))

  ⟦_⟧ᵗ : ∀ {Γ : Context} {T : Type Γ}
    → Term T
    → (⟦Γ⟧ : ⟦ Γ ⟧ᶜ)
    → ⟦ T ⟧ᵀ ⟦Γ⟧
  ⟦ ⌜ x ⌝ᵀ ⟧ᵗ ⟦Γ⟧ = lift x
  ⟦ ⌜ x ⌝ᵗ ⟧ᵗ ⟦Γ⟧ = lift x
  ⟦ ‘⌜‘VAR₀’⌝ᵗ’ ⟧ᵗ ⟦Γ⟧
    = lift ⌜ lower (Σ.snd ⟦Γ⟧) ⌝ᵗ
  ⟦ ‘⌜‘VAR₀’⌝ᵀ’ ⟧ᵗ ⟦Γ⟧
    = lift ⌜ lower (Σ.snd ⟦Γ⟧) ⌝ᵀ
  ⟦ f ‘’ₐ x ⟧ᵗ ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧ (⟦ x ⟧ᵗ ⟦Γ⟧)
  ⟦ ‘tt’ ⟧ᵗ ⟦Γ⟧ = tt
  ⟦ quine→ {φ} ⟧ᵗ ⟦Γ⟧ x = x
  ⟦ quine← {φ} ⟧ᵗ ⟦Γ⟧ x = x
  ⟦ ‘λ’ f ⟧ᵗ ⟦Γ⟧ x = ⟦ f ⟧ᵗ (⟦Γ⟧ , x)
  ⟦ ‘VAR₀’ ⟧ᵗ ⟦Γ⟧ = Σ.snd ⟦Γ⟧
  ⟦ SW t ⟧ᵗ = ⟦ t ⟧ᵗ
  ⟦ ←SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
  ⟦ →SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
  ⟦ ←SW₁SV→SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
  ⟦ →SW₁SV→SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
  ⟦ w x ⟧ᵗ ⟦Γ⟧ = ⟦ x ⟧ᵗ (Σ.fst ⟦Γ⟧)
  ⟦ w→ f ⟧ᵗ ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦ →w f ⟧ᵗ ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦ ww→ f ⟧ᵗ ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦ →ww f ⟧ᵗ ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦ ‘‘×’’’ ⟧ᵗ ⟦Γ⟧ A B = lift (lower A ‘×’ lower B)
```

```
  ⟦ g `∘` f ⟧ᵗ ⟦Γ⟧ x = ⟦ g ⟧ᵗ ⟦Γ⟧ (⟦ f ⟧ᵗ ⟦Γ⟧ x)
  ⟦ f w``ₐ x ⟧ᵗ ⟦Γ⟧
    = lift (lower (⟦ f ⟧ᵗ ⟦Γ⟧) ``ₐ lower (⟦ x ⟧ᵗ ⟦Γ⟧))
  ⟦ ``ₐ ⟧ᵗ ⟦Γ⟧ f x
    = lift (lower f ``ₐ lower x)
  ⟦ ``□`` {Γ} T ⟧ᵗ ⟦Γ⟧
    = lift (`Term` `` lower (⟦ T ⟧ᵗ ⟦Γ⟧))
  ⟦ A ``→`` B ⟧ᵗ ⟦Γ⟧
    = lift
        (lower (⟦ A ⟧ᵗ ⟦Γ⟧) `→` lower (⟦ B ⟧ᵗ ⟦Γ⟧))
  ⟦ A ```→``` B ⟧ᵗ ⟦Γ⟧
    = lift
        (lower (⟦ A ⟧ᵗ ⟦Γ⟧) ``→`` lower (⟦ B ⟧ᵗ ⟦Γ⟧))
  ⟦ A ```×``` B ⟧ᵗ ⟦Γ⟧
    = lift
        (lower (⟦ A ⟧ᵗ ⟦Γ⟧) ``×`` lower (⟦ B ⟧ᵗ ⟦Γ⟧))


module inner (`X` : Type ε)
  (`f` : Term {ε} (`□` `X` `→` `X`))
where
  `H` : Type ε
  `H` = Quine (W₁ `Term` `` `VAR₀` `→` W `X`)

  `toH` : □ ((`□` `H` `→` `X`) `→` `H`)
  `toH` = ←SW₁SV→W quine←

  `fromH` : □ (`H` `→` (`□` `H` `→` `X`))
  `fromH` = →SW₁SV→W quine→

  `□H→□X` : □ (`□` `H` `→` `□` `X`)
  `□H→□X`
    = `λ` (w ⌜ `fromH` ⌝ᵗ
      w``ₐ `VAR₀`
      w``ₐ ⌜`VAR₀`⌝ᵗ)

  `h` : Term `H`
  `h` = `toH` ``ₐ (`f` `∘` `□H→□X`)

  Löb : □ `X`
  Löb = `fromH` ``ₐ `h` ``ₐ ⌜ `h` ⌝ᵗ

Löb : ∀ {X}
  → Term {ε} (`□` X `→` X) → Term {ε} X
Löb {X} f = inner.Löb X f

⟦_⟧ : Type ε → Set _
⟦ T ⟧ = ⟦ T ⟧ᵀ tt

`¬` _ : ∀ {Γ} → Type Γ → Type Γ
`¬` T = T `→` `⊥`

_w``×``_ : ∀ {Γ X}
  → Term {Γ ▷ X} (W (`Type` Γ))
  → Term {Γ ▷ X} (W (`Type` Γ))
  → Term {Γ ▷ X} (W (`Type` Γ))
A w``×`` B = w→ (w→ (w ``×``) ``ₐ A) ``ₐ B

löb : ∀ {`X`} → □ (`□` `X` `→` `X`) → ⟦ `X` ⟧
löb f = ⟦ Löb f ⟧ᵗ tt

incompleteness : ¬ □ (`¬` (`□` `⊥`))
incompleteness = löb

soundness : ¬ □ `⊥`
soundness x = ⟦ x ⟧ᵗ tt

non-emptyness : Σ (Type ε) (λ T → □ T)
non-emptyness = `⊤` , `tt`
```

C. Proving that FairBot Cooperates with Itself

We begin with the definitions of a few particularly useful dependent combinators:

```
_∘_ : ∀ {A : Set}
  {B : A → Set}
  {C : {x : A} → B x → Set}
  → ({x : A} (y : B x) → C y)
  → (g : (x : A) → B x) (x : A)
  → C (g x)
f ∘ g = λ x → f (g x)

infixl 8 _ˢ_

_ˢ_ : ∀ {A : Set}
  {B : A → Set}
  {C : (x : A) → B x → Set}
  → ((x : A) (y : B x) → C x y)
  → (g : (x : A) → B x) (x : A)
  → C x (g x)
f ˢ g = λ x → f x (g x)

ᵏ : {A B : Set} → A → B → A
ᵏ a b = a

^ : ∀ {S : Set} {T : S → Set} {P : Σ S T → Set}
    → ((σ : Σ S T) → P σ)
    → (s : S) (t : T s) → P (s , t)
^ f s t = f (s , t)
```

It turns out that we can define all the things we need for proving self-cooperation of FairBot in a variant of the simply typed lambda calculus (STLC). In order to do this, we do not index types over contexts. Rather than using `Term {Γ} T`, we will denote the type of terms in context Γ of type `T` as `Γ ⊢ T`, the standard notation for "provability". Since our types are no longer indexed over contexts, we can represent a context as a list of types.

```
infixr 5 _⊢_
infixr 10 _`→`_ _`×`_

data Type : Set where
  _`⊢`_ : List Type → Type → Type
  _`→`_ _`×`_ : Type → Type → Type
  `⊥` `⊤` : Type

Context = List Type
```

We will then need some way to handle binding. For simplicity, we'll make use of a dependent form of DeBrujin variables.

```
data _∈_ (T : Type) : Context → Set where
```

First we want our "variable zero", which lets us pick off the "top" element of the context.

```
  top : ∀ {Γ} → T ∈ (T :: Γ)
```

Then we want a way to extend variables to work in larger contexts.

```
  pop : ∀ {Γ S} → T ∈ Γ → T ∈ (S :: Γ)
```

And, finally, we are ready to define the term language for our extended STLC.

```
data _⊢_ (Γ : Context) : Type → Set where
```

The next few constructors are fairly standard. Before anything else, we want to be able to lift bindings into terms.

```
  var : ∀ {T} → T ∈ Γ → Γ ⊢ T
```

Then the intro rules for all of our easier datatypes.

```
  <> : Γ ⊢ '⊤'
  _,_ : ∀ {A B} → Γ ⊢ A → Γ ⊢ B → Γ ⊢ A '×' B
  '⊥'-elim : ∀ {A} → Γ ⊢ '⊥' → Γ ⊢ A
  π₁ : ∀ {A B} → Γ ⊢ A '×' B → Γ ⊢ A
  π₂ : ∀ {A B} → Γ ⊢ A '×' B → Γ ⊢ B
  'λ' : ∀ {A B} → (A :: Γ) ⊢ B → Γ ⊢ (A '→' B)
  _''ₐ_ : ∀ {A B} → Γ ⊢ (A '→' B) → Γ ⊢ A → Γ ⊢ B
```

At this point things become more delicate. To properly capture Gödel–Löb modal logic, abbreviated as GL, we want our theory to validate the rules

1. $\vdash$ `A` $\rightarrow$ $\vdash$ $\Box$ `A`
2. $\vdash$ $\Box$ `A ‘→’ □ □ A`

However, it should *not* validate $\vdash$ `A ‘→’ □ A`. If we only had the unary $\Box$ operator we would run into difficulty later. Crucially, we couldn't add the rule $\Gamma \vdash$ `A` $\rightarrow$ $\Gamma \vdash \Box$ `A`, since this would let us prove `A ‘→’ □ A`.

We will use Gödel quotes to denote the constructor corresponding to rule 1:

```
  ⌜_⌝ : ∀ {Δ A} → Δ ⊢ A → Γ ⊢ (Δ '⊢' A)
```

Similarly, we will write the rule validating `□ A ‘→’ □ □ A` as `repr`.

```
  repr : ∀ {Δ A} → Γ ⊢ (Δ '⊢' A) → Γ ⊢ (Δ '⊢' (Δ '⊢' A))
```

We would like to be able to apply functions under $\Box$, and for this we introduce the so-called "distribution" rule. In GL, it takes the form $\vdash \Box$ `(A ‘→’ B)` $\rightarrow \vdash$ `(□ A ‘→’ □ B)`. For us it is not much more complicated.

```
  dist : ∀ {Δ A B}
    → Γ ⊢ (Δ '⊢' (A '→' B))
    → Γ ⊢ (Δ '⊢' A)
    → Γ ⊢ (Δ '⊢' B)
```

And, finally, we include the Löbian axiom.

```
  Löb : ∀ {Δ A}
    → Γ ⊢ (Δ '⊢' ((Δ '⊢' A) '→' A))
    → Γ ⊢ (Δ '⊢' A)
```

From these constructors we can prove the simpler form of the Löb rule.

```
löb : ∀ {Γ A} → Γ ⊢ ((Γ '⊢' A) '→' A) → Γ ⊢ A
löb t = t ''ₐ Löb ⌜ t ⌝
```

Of course, because we are using DeBrujin indices, before we can do too much we'll need to give an account of lifting. Thankfully, unlike when we were dealing with dependent type theory, we can define these computationally, and get for free all the congruences we had to add as axioms before.

Our definition of weakening is unremarkable, and sufficiently simple that Agsy, Agda's automatic proof-finder, was able to fill in all of the code; we include it in the artifact and elide all but the type signature from the paper.

```
lift-tm
  : ∀ {Γ A} T Δ → (Δ ++ Γ) ⊢ A → (Δ ++ (T :: Γ)) ⊢ A
```

Weakening is a special case of `lift-tm`.

```
wk : ∀ {Γ A B} → Γ ⊢ A → (B :: Γ) ⊢ A
wk = lift-tm _ ε
```

Finally, we define function composition for our internal language.

```
infixl 10 _∘'_
_∘'_ : ∀ {Γ A B C}
  → Γ ⊢ (B '→' C)
  → Γ ⊢ (A '→' B)
  → Γ ⊢ (A '→' C)
f ∘' g = 'λ' (wk f ''ₐ (wk g ''ₐ var top))
```

Now we are ready to prove that FairBot cooperates with itself. Sadly, our type system isn't expressive enough to give a general type of bots, but we can still prove things about the interactions of particular bots if we substitute their types by hand. For example, we can state the desired theorem (that FairBot cooperates with itself) as:

```
distf : ∀ {Γ Δ A B}
  → Γ ⊢ (Δ '⊢' A '→' B)
  → Γ ⊢ (Δ '⊢' A) '→' (Δ '⊢' B)
distf bf = 'λ' (dist (wk bf) (var top))

evf : ∀ {Γ Δ A}
  → Γ ⊢ (Δ '⊢' A) '→' (Δ '⊢' (Δ '⊢' A))
evf = 'λ' (repr (var top))

fb-fb-cooperate : ∀ {Γ A B}
  → Γ ⊢ (Γ '⊢' A) '→' B
  → Γ ⊢ (Γ '⊢' B) '→' A
  → Γ ⊢ (A '×' B)
fb-fb-cooperate a b
  = löb (b ∘' distf ⌜ a ⌝ ∘' evf)
  , löb (a ∘' distf ⌜ b ⌝ ∘' evf)
```

We can also state the theorem in a more familiar form with a couple abbreviations

```
'□' = _'⊢'_ ε
□ = _⊢_ ε

fb-fb-cooperate' : ∀ {A B}
  → □ ('□' A '→' B)
  → □ ('□' B '→' A)
  → □ (A '×' B)
fb-fb-cooperate' = fb-fb-cooperate
```

In the file `fair-bot-self-cooperates.lagda` in the artifact, we show all the meta-theoretic properties we had before: soundness, inhabitedness, and incompleteness.

Acknowledgments

[Redacted for the anonymous submission]

References

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 57–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-677-6. doi: 10.1145/1292597.1292608. URL `http://www.cs.nott.ac.uk/~psztxa/publ/obseqnow.pdf`.

A. W. Appel, P.-A. Mellies, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. *ACM SIGPLAN Notices*, 42(1):109–122, 2007. URL `https://www.cs.princeton.edu/~appel/papers/modalmodel.pdf`.

M. Barasz, P. Christiano, B. Fallenstein, M. Herreshoff, P. LaVictoire, and E. Yudkowsky. Robust cooperation in the prisoner's dilemma: Program equilibrium via provability logic. *ArXiv e-prints*, Jan 2014. URL `http://arxiv.org/pdf/1401.5577v1.pdf`.

M. Brown and J. Palsberg. Breaking through the normalization barrier: A self-interpreter for f-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 5–17. ACM, 2016. doi: 10.1145/2837614.2837623. URL `http://compilers.cs.ucla.edu/popl16/popl16-full.pdf`.

J. Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2008.12.114. URL http://www.sciencedirect.com/science/article/pii/S157106610800577X. Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, chapter A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family, pages 93–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74464-1. doi: 10.1007/978-3-540-74464-1_7. URL http://dx.doi.org/10.1007/978-3-540-74464-1_7.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931. URL http://www.w-k-essler.de/pdfs/goedel.pdf.

D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Vintage, 1979. ISBN 978-0394745022.

S. C. Kleene. *Introduction to Metamathematics*. Wolters-Noordhoff, 1952. ISBN 0-7204-2103-9.

C. McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 1–12. ACM, 2010. URL https://personal.cis.strath.ac.uk/conor.mcbride/pub/DepRep/DepRep.pdf.

R. O'Connor. Essential incompleteness of arithmetic verified by coq. *CoRR*, abs/cs/0505034, 2005. URL http://arxiv.org/abs/cs/0505034.

L. C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *Journal of Automated Reasoning*, 55 (1):1–37, 2015. URL https://www.cl.cam.ac.uk/~lp15/papers/Formath/Goedel-ar.pdf.

D. Piponi. From löb's theorem to spreadsheet evaluation, November 2006. URL http://blog.sigfpe.com/2006/11/from-l-theorem-to-spreadsheet.html.

G. K. Pullum. Scooping the loop snooper, October 2000. URL http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html.

N. Shankar. *Proof-checking Metamathematics (Theorem-proving)*. PhD thesis, The University of Texas at Austin, 1986. AAI8717580.

N. Shankar. *Metamathematics, Machines and Gödel's Proof*. Cambridge University Press, 1997.

R. J. Simmons and B. Toninho. Constructive provability logic. *CoRR*, abs/1205.6402, May 2012. URL http://arxiv.org/abs/1205.6402.

A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936. URL http://www.w-k-essler.de/pdfs/Tarski.pdf.

B. Yudkowsky. Lob's theorem cured my social anxiety, February 2014. URL http://agentyduck.blogspot.com/2014/02/lobs-theorem-cured-my-social-anxiety.html.