Löb's Theorem

A functional pearl of dependently typed quining

Jason Gross
MIT CSAIL
jgross@mit.edu

Name2 Name3
Affiliation2/3
Email2/3

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms Agda, Lob, quine, self-reference

Keywords Agda, Lob, quine, self-reference

Abstract

This is the text of the abstract.

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* (Pullum 2000))

TODO

- cite Using Reflection to Explain and Enhance Type Theory?

1. Introduction

Löb's thereom has a variety of applications, from proving incompleteness of a logical theory as a trivial corrolary, to acting as a no-go theorem for a large class of self-interpreters (TODO: mention F_{omega}?), from allowing robust cooperation in the Prisoner's Dilemma with Source Code (), to curing social anxiety ().

"What is Löb's theorem, this versatile tool with wonderous 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, dear reader are the most awesome person in the world. Since this is exactly what the sentence asserts, the sentence is true, and you, dear reader, 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 $A \to B$ are the same, if we assume A, we are also assuming $A \to B$, and hence we have B, and since assuming A yields B, we have that $A \to B$. 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.

Have you figured out what went wrong?

The sentence that we have been considering is not a valid mathematical sentence. Ask yourself what makes it invalid, while we consider a similar sentence that is actually valid.

Now consider the sentence"if this sentence is provable, then you, dear reader, are the most awesome person in the world." Fix a particular formalization of provability (for example, Peano Arithmetic, or Martin–Löf Type Theory). To prove that this sentence is true, suppose that it is provable. We must now show that you, dear reader, are the most awesome person in the world. *If provability implies truth*, then the sentence is true, and then you, dear reader, are the most awesome person in the world. Thus, 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. Symbolically, this is

$$\Box(\Box X- > X) \to \Box X$$

where $\Box X$ means "X is provable" (in our fixed formalization of provability).

Let us now return to the question we posed above: what went wrong with our original sentence? The answer is that self-reference with truth is impossible, and the clearest way I 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

[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 counterfactual "if".

Logic	Programming	Set Theory
Proposition	Type	Set of Proofs
Proof	Program	Element
Implication ($\rightarrow$)	Function ($\rightarrow$)	Function
Conjunction ($\wedge$)	Pairing (,)	Cartesian Product ($\times$)
Disjunction ($\vee$)	Sum (+)	Disjoint Union ($\sqcup$)
Gödel codes	ASTs	—

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

proofs, a more natural choice of encoding programs will be abstract syntax trees. 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.

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

Before diving into Löb's theorem in detail, we'll first visit a standard paradigm for formalizing the syntax of dependent type theory. (TODO: Move this?)

2. Quines

What is the computational equivalent of the sentence "If this sentence is provable, then X"? It will be something of the form "??? $\rightarrow X$". As a warm-up, let's look at a Python program that returns a string representation of this type.

To do this, we need a 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 (or sassy?). These "cheating" solutions are:

- The empty program, which outputs nothing.
- The program `print(open(__file__, 'r').read())`, which relies on the Python interpreter to get the source code of the program.

Now we develop the standard solution. At a first gloss, it looks like:

```
(lambda T: '(' + T + ') -> X') "???"
```

Now we need to replace `"???"` with the entirety of this program code. We use Python's string escaping function (`repr`) and replacement syntax (`("foo %s bar" % "baz")` becomes `"foo baz bar"`):

```
(lambda T: '(' + T % repr(T) + ') → X')
 ("(lambda T: '(' + T %% repr(T) + ') → X')\n (%s)")
```

This is a slight modification on the standard way of programming a quine, a program that outputs its own source-code.

Suppose we have a function □ that takes in a string representation of a type, and returns the type of syntax trees of programs producing that type. Then our Löbian sentence would look something like (if → were valid notation for function types in Python)

```
(lambda T: □ (T % repr(T)) → X)
 ("(lambda T: □ (T %% repr(T)) → X)\n (%s)")
```

Now, finally, we can see what goes wrong when we consider using "if this sentence is true" rather than "if this sentence is provable". Provability corresponds to syntax trees for programs; truth corresponds to execution of the program itself. Our pseudo-Python thus becomes

```
(lambda T: eval(T % repr(T)) → X)
 ("(lambda T: eval(T %% repr(T)) → X)\n (%s)")
```

This code never terminates! So, in a quite literal sense, the issue with our original sentence was that, if we tried to phrase it, we'd never finish.

Note well that the type (□ X → X) is a type that takes syntax trees and evaluates them; it is the type of an interpreter. (TODO: maybe move this sentence?)

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. (TODO: citations) For example, here is a very simple (and incomplete) formalization with Π, a unit type ($\top$), an empty type ($\bot$), and lambdas. (TODO: FIXME: What's the right level of simplicity?) TODO: mention convention of ''?

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 {Γ ▷ A} B → Term ('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. Here is an example of such a function:

mutual

$$
\begin{array}{l}
\llbracket\_\rrbracket^{c} : \mathsf{Context} \rightarrow \mathsf{Set} \\
\llbracket\, \varepsilon \,\rrbracket^{c} = \top \\
\llbracket\, \Gamma \triangleright T \,\rrbracket^{c} = \Sigma\, \llbracket\, \Gamma \,\rrbracket^{c}\, \llbracket\, T \,\rrbracket^{T} \\
\\
\llbracket\_\rrbracket^{T} : \forall \{\Gamma\} \rightarrow \mathsf{Type}\ \Gamma \rightarrow \llbracket\, \Gamma \,\rrbracket^{c} \rightarrow \mathsf{Set} \\
\llbracket\, `\top\text{'} \,\rrbracket^{T}\, \llbracket\Gamma\rrbracket = \top \\
\llbracket\, `\bot\text{'} \,\rrbracket^{T}\, \llbracket\Gamma\rrbracket = \bot \\
\llbracket\, `\Pi\text{'}\ A\ B \,\rrbracket^{T}\, \llbracket\Gamma\rrbracket = (x : \llbracket\, A \,\rrbracket^{T}\, \llbracket\Gamma\rrbracket) \rightarrow \llbracket\, B \,\rrbracket^{T}\, (\llbracket\Gamma\rrbracket\ ,\ x) \\
\\
\llbracket\_\rrbracket^{t} : \forall \{\Gamma\ T\} \rightarrow \mathsf{Term}\ \{\Gamma\}\ T \rightarrow (\llbracket\Gamma\rrbracket : \llbracket\, \Gamma \,\rrbracket^{c}) \rightarrow \llbracket\, T \,\rrbracket^{T}\, \llbracket\Gamma\rrbracket \\
\llbracket\, `\mathsf{tt}\text{'} \,\rrbracket^{t}\, \llbracket\Gamma\rrbracket = \mathsf{tt} \\
\llbracket\, `\lambda\text{'}\ f \,\rrbracket^{t}\, \llbracket\Gamma\rrbracket\ x = \llbracket\, f \,\rrbracket^{t}\, (\llbracket\Gamma\rrbracket\ ,\ x)
\end{array}
$$

TODO: Maybe mention something about the denotation function being "local", i.e., not needing to do anything but the top-level case-analysis?

4. This Paper

In this paper, we make extensive use of this trick for validating models. 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 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 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()`. Finally, we sketch our implementation of Löb's theorem (code in an appendix) based on the assumption only 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 with an application to the prisoner's dilemma, as well as some discussion about avenues for removing the hard-coded `repr`. TODO: Ensure that this ordering is accurate

5. Prior Work

TODO: Use of Löb's theorem in program logic as an induction principle? (TODO)

TODO: Brief mention of Lob's theorem in Haskell / elsewhere / ? (TODO)

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 `Term`s 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` TODO: This is probably unclear. Maybe mention repr?.

```
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 existance of an self-interpreter for F_ω (Brown and Palsberg 2016).[2]

[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.

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 disucss 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-Emptyness

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

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

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

¬_ : Set → Set
¬ T = T → ⊥

'¬'_ : Type → Type
'¬' T = T '→' '⊥'

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

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

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

non-emptyness : □ '⊤'
non-emptyness = 'tt'

no-interpreters : ¬ (∀ {'X'} → □ ('□' 'X' '→' 'X'))
no-interpreters interp = löb (interp {'⊥'})
```

TODO: Does this code need any explanation? Maybe for no-interpreters?

8. Encoding with Quines

```agda
module lob-by-quines where
infixl 2 _▻_
infixl 3 _‘’_
infixr 1 _‘→’_
infixl 3 _‘’ₐ_
infixl 3 _w‘‘’’ₐ_
infixr 2 _‘∘’_

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

  data Type : Context → Set where
    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
    ‘Typeε’ : ∀ {Γ} → Type Γ
    ‘□’ : ∀ {Γ} → Type (Γ ▻ ‘Typeε’)
    _‘→’_ : ∀ {Γ} → Type Γ → Type Γ → Type Γ
    Quine : Type (ε ▻ ‘Typeε’) → Type ε
    ‘⊤’ : ∀ {Γ} → Type Γ
    ‘⊥’ : ∀ {Γ} → Type Γ

  data Term : {Γ : Context} → Type Γ → Set where
    ⌜_⌝ᵀ : ∀ {Γ} → Type ε → Term {Γ} ‘Typeε’
    ⌜_⌝ᵗ : ∀ {Γ T} → Term {ε} T → Term {Γ} (‘□’ ‘’ ⌜ T ⌝ᵀ)
    ‘⌜‘VAR₀’⌝ᵗ’ : ∀ {T}
      → Term {ε ▻ ‘□’ ‘’ ⌜ T ⌝ᵀ} (W (‘□’ ‘’ ⌜ ‘□’ ‘’ ⌜ T ⌝ᵀ ⌝ᵀ))
    ‘λ•’ : ∀ {Γ A B} → Term {Γ ▻ A} (W B) → Term (A ‘→’ B)
    ‘VAR₀’ : ∀ {Γ T} → Term {Γ ▻ T} (W T)
    _‘’ₐ_ : ∀ {Γ A B}
      → Term {Γ} (A ‘→’ B)
      → Term {Γ} A
      → Term {Γ} B
    quine→ : ∀ {ϕ} → Term {ε} (Quine ϕ ‘→’ ϕ ‘’ ⌜ Quine ϕ ⌝ᵀ)
    quine← : ∀ {ϕ} → Term {ε} (ϕ ‘’ ⌜ Quine ϕ ⌝ᵀ ‘→’ Quine ϕ)
    ‘tt’ : ∀ {Γ} → Term {Γ} ‘⊤’
    →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)
    w : ∀ {Γ A T} → Term {Γ} A → Term {Γ ▻ T} (W A)
    w→ : ∀ {Γ A B X}
      → Term {Γ} (A ‘→’ B)
      → Term {Γ ▻ X} (W A ‘→’ W B)
    _‘∘’_ : ∀ {Γ A B C}
      → Term {Γ} (B ‘→’ C)
      → Term {Γ} (A ‘→’ B)
      → Term {Γ} (A ‘→’ C)
    _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}

max-level : Level
max-level = lzero -- also works for any higher level

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
  ⟦ W T ⟧ᵀ ⟦Γ⟧ = ⟦ T ⟧ᵀ (Σ.proj₁ ⟦Γ⟧)
  ⟦ W₁ T ⟧ᵀ ⟦Γ⟧ = ⟦ T ⟧ᵀ ((Σ.proj₁ (Σ.proj₁ ⟦Γ⟧)) , (Σ.proj₂ ⟦Γ⟧))
  ⟦ T ‘’ x ⟧ᵀ ⟦Γ⟧ = ⟦ T ⟧ᵀ (⟦Γ⟧ , ⟦ x ⟧ᵗ ⟦Γ⟧)
  ⟦ ‘Typeε’ ⟧ᵀ ⟦Γ⟧ = Lifted (Type ε)
  ⟦ ‘□’ ⟧ᵀ ⟦Γ⟧ = Lifted (Term {ε} (lower (Σ.proj₂ ⟦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 T} → Term {Γ} 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.proj₂ ⟦Γ⟧) ⌝ᵗ
  ⟦ f ‘’ₐ x ⟧ᵗ ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧ (⟦ x ⟧ᵗ ⟦Γ⟧)
  ⟦ ‘tt’ ⟧ᵗ ⟦Γ⟧ = tt
  ⟦ quine→ ⟧ᵗ ⟦Γ⟧ x = x
  ⟦ quine← ⟧ᵗ ⟦Γ⟧ x = x
  ⟦ ‘λ•’ f ⟧ᵗ ⟦Γ⟧ x = ⟦ f ⟧ᵗ (⟦Γ⟧ , x)
  ⟦ ‘VAR₀’ ⟧ᵗ ⟦Γ⟧ = Σ.proj₂ ⟦Γ⟧
  ⟦ ←SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
  ⟦ →SW₁SV→W f ⟧ᵗ = ⟦ f ⟧ᵗ
  ⟦ w x ⟧ᵗ ⟦Γ⟧ = ⟦ x ⟧ᵗ (Σ.proj₁ ⟦Γ⟧)
  ⟦ w→ f ⟧ᵗ ⟦Γ⟧ = ⟦ f ⟧ᵗ (Σ.proj₁ ⟦Γ⟧)
  ⟦ 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⟧))

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

¬_ : ∀ {ℓ m} → Set ℓ → Set (ℓ ⊔ m)
¬_ {ℓ} {m} T = T → ⊥ {m}

incompleteness : ¬ □ (‘¬’ (‘□’ ‘’ ⌜ ‘⊥’ ⌝ᵀ))
incompleteness = löb

soundness : ¬ □ ‘⊥’
soundness x = ⟦ x ⟧ᵗ tt

non-emptyness : Σ (Type ε) (λ T → □ T)
non-emptyness = ‘⊤’ , ‘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.

```
open lob

-- a bot takes in the source code for itself,
-- for another bot, and spits out the assertion
-- that it cooperates with this bot
‘Bot’ : ∀ {Γ} → Type Γ
‘Bot’ {Γ}
  = Quine (W₁ ‘Term’ ‘’ ‘VAR₀’
    ‘→’ W₁ ‘Term’ ‘’ ‘VAR₀’
    ‘→’ W (‘Type’ Γ))

_cooperates-with_ : □ ‘Bot’ → □ ‘Bot’ → Type ε
b₁ cooperates-with b₂ = lower (⟦ b₁ ⟧ᵗ tt (lift b₁) (lift b₂))

‘eval-bot’’ : ∀ {Γ} → Term {Γ} (‘Bot’ ‘→’ (‘□’ ‘Bot’ ‘→’ ‘□’ ‘Bot’ ‘→’ ‘Type’ Γ))
‘eval-bot’’ = →SW₁SV→SW₁SV→W quine→

‘‘eval-bot’’’ : ∀ {Γ} → Term {Γ} (‘□’ ‘Bot’ ‘→’ ‘□’ ({- other -} ‘□’ ‘Bot’ ‘→’ ‘Type’ Γ))
‘‘eval-bot’’’ = ‘λ•’ (w ⌜ ‘eval-bot’’ ⌝ᵗ w‘‘’’ₐ ‘VAR₀’ w‘‘’’ₐ ‘⌜‘VAR₀’ ...

‘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’))))
  where
    ‘eval-other’ : Term {Γ ▹ ‘□’ ‘Bot’ ▹ W (‘□’ ‘Bot’)} (W (W (‘□’ (‘□’ ‘Bot’ ‘→’ ‘Type’ Γ))))
    ‘eval-other’ = w→ (w (w→ (w ‘‘eval-bot’’’))) ‘’ₐ ‘VAR₀’

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

‘self’ : ∀ {Γ} → Term {Γ ▹ ‘□’ ‘Bot’ ▹ W (‘□’ ‘Bot’)} (W (W (‘□’ ‘Bot’)))
‘self’ = w ‘VAR₀’

‘other’ : ∀ {Γ} → Term {Γ ▹ ‘□’ ‘Bot’ ▹ W (‘□’ ‘Bot’)} (W (W (‘□’ ‘Bot’)))
‘other’ = ‘VAR₀’
```

```
make-bot : ∀ {Γ} → Term {Γ ▹ ‘□’ ‘Bot’ ▹ W (‘□’ ‘Bot’)} (W (W (‘Type’ ...
make-bot t = ←SW₁SV→SW₁SV→W quine← ‘’ₐ ‘λ•’ (→w (‘λ•’ t))

ww‘‘‘¬’’’_ : ∀ {Γ A B}
  → Term {Γ ▹ A ▹ B} (W (W (‘□’ (‘Type’ Γ))))
  → Term {Γ ▹ A ▹ B} (W (W (‘□’ (‘Type’ Γ))))
ww‘‘‘¬’’’ T = T ww‘‘‘→’’’ w (w ⌜ ⌜ ‘⊥’ ⌝ ⌝ᵗ)

‘DefectBot’ : □ ‘Bot’
‘CooperateBot’ : □ ‘Bot’
‘FairBot’ : □ ‘Bot’
‘PrudentBot’ : □ ‘Bot’

‘DefectBot’ = make-bot (w (w ⌜ ‘⊥’ ⌝))
‘CooperateBot’ = make-bot (w (w ⌜ ‘⊤’ ⌝))
‘FairBot’ = make-bot (‘‘□’’ (‘other-cooperates-with’ ‘’ₐ ‘self’))
‘PrudentBot’ = make-bot (‘‘□’’ (ϕ₀ ww‘‘‘×’’’ (¬□⊥ ww‘‘‘→’’’ other-defe...
  where
    ϕ₀ : ∀ {Γ} → Term {Γ ▹ ‘□’ ‘Bot’ ▹ W (‘□’ ‘Bot’)} (W (W (‘□’ (‘Ty...
    ϕ₀ = ‘other-cooperates-with’ ‘’ₐ ‘self’

    other-defects-against-DefectBot : Term {_ ▹ ‘□’ ‘Bot’ ▹ W (‘□’ ‘Bot...
    other-defects-against-DefectBot = ww‘‘‘¬’’’ (‘other-cooperates-with...

    ¬□⊥ : ∀ {Γ A B} → Term {Γ ▹ A ▹ B} (W (W (‘□’ (‘Type’ Γ))))
    ¬□⊥ = w (w ⌜ ⌜ ‘¬’ (‘□’ ‘⊥’) ⌝ ⌝ᵗ)
```

10. Encoding with Add-Quote Function

(appendix) - Discuss whiteboard phrasing of sentence with sigmas - It remains to show that we can construct - Discuss whiteboard phrasing of untyped sentence - Given: - X - □ = Term - f : □ ’X’ -> X - define y : X - Suppose we have a type H ≅ Term ⌜ H → X ⌝, and we have - toH : Term ⌜ H → X ⌝ → H - fromH : H → Term ⌜ H → X ⌝ - quote : H → Term ⌜ H ⌝ - - Then we can define - `y = (λ h : H. f (subst (quote h) h) (toH '\h : H. f (subst (quo...`

11. Removing add-quote and actually tying the knot (future work 1)

- Temporary outline section to be moved

-

- How do we construct the Curry–Howard analogue of the Löbian sentence? A quine is a program that outputs its own source code (). We will say that a *type-theoretic quine* is a program that outputs its own (well-typed) abstract syntax tree. Generalizing this slightly, we can consider programs that output an arbitrary function of their own syntax trees.

- TODO: Examples of double quotation, single quotation, etc.

- Given any function ϕ from doubly-quoted syntactic types to singly-quoted syntactic types, and given an operator ⌜_⌝ which adds an extra level of quotation, we can define the type of a *quine at* ϕ to be a (syntactic) type "Quine ϕ" which is isomorphic to "ϕ (⌜Quine ϕ⌝)".

- What's wrong is that self-reference with truth is impossible. In a particular technical sense, it doesn't terminate. Solution: Provability

- Quining / self-referential provability sentence and provability implies truth

- Curry–Howard, quines, abstract syntax trees (This is an interpreter!)

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

record Σ {a p} (A : Set a) (P : A → Set p) : Set (a ⊔ p) where
  constructor _,_
  field
    proj₁ : A
    proj₂ : P proj₁

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

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
```

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

```
module lob where
  infixl 2 _▷_
  infixl 3 _‘’_
  infixr 1 _‘→’_
  infixr 1 _‘‘→’’_
  infixr 1 _ww‘‘‘→’’’_
  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
      W : ∀ {Γ A} → Type Γ → Type (Γ ▷ A)
      W₁ : ∀ {Γ A B} → Type (Γ ▷ B) → Type (Γ ▷ A ▷ (W {Γ = Γ} {A
      _‘’_ : ∀ {Γ A} → Type (Γ ▷ A) → Term {Γ} A → Type Γ
      ‘Type’ : ∀ Γ → Type Γ
      ‘Term’ : ∀ {Γ} → Type (Γ ▷ ‘Type’ Γ)
      _‘→’_ : ∀ {Γ} → Type Γ → Type Γ → Type Γ
      _‘×’_ : ∀ {Γ} → Type Γ → Type Γ → Type Γ
      Quine : ∀ {Γ} → Type (Γ ▷ ‘Type’ Γ) → Type Γ
      ‘⊤’ : ∀ {Γ} → Type Γ
      ‘⊥’ : ∀ {Γ} → Type Γ

    data Term : {Γ : Context} → Type Γ → Set where
      ⌜_⌝ : ∀ {Γ} → Type Γ → Term {Γ} (‘Type’ Γ)
      ⌜_⌝ᵗ : ∀ {Γ T} → Term {Γ} T → Term {Γ} (‘Term’ ‘’ ⌜ T ⌝)
      ‘⌜‘VAR₀’⌝ᵗ’ : ∀ {Γ T} → Term {Γ ▷ ‘Term’ ‘’ ⌜ T ⌝} (W (‘Term’ ‘’ ⌜
      ‘⌜‘VAR₀’⌝’ : ∀ {Γ} → Term {Γ ▷ ‘Type’ Γ} (W (‘Term’ ‘’ ⌜ ‘Type’
      ‘λ●’ : ∀ {Γ A B} → Term {Γ ▷ A} (W B) → Term {Γ} (A ‘→’ B)
      ‘VAR₀’ : ∀ {Γ T} → Term {Γ ▷ T} (W T)
      _‘’ₐ_ : ∀ {Γ A B} → Term {Γ} (A ‘→’ B) → Term {Γ} A → Term
      ‘‘×’’’ : ∀ {Γ} → Term {Γ} (‘Type’ Γ ‘→’ ‘Type’ Γ ‘→’ ‘Type’ Γ)
      quine→ : ∀ {Γ ϕ} → Term {Γ} (Quine ϕ ‘→’ ϕ ‘’ ⌜ Quine ϕ ⌝)
      quine← : ∀ {Γ ϕ} → Term {Γ} (ϕ ‘’ ⌜ Quine ϕ ⌝ ‘→’ Quine ϕ)
      ‘tt’ : ∀ {Γ} → Term {Γ} ‘⊤’
      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
        → 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
        → 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 : ∀ {Γ A B X} → Term {Γ ▷ X} (W A ‘→’ W B) → Term {Γ ▷
      ww→ : ∀ {Γ A B X Y} → Term {Γ ▷ X ▷ Y} (W (W (A ‘→’ B))) →
      →ww : ∀ {Γ A B X Y} → Term {Γ ▷ X ▷ Y} (W (W A) ‘→’ W (W B
      _‘∘’_ : ∀ {Γ A B C} → Term {Γ} (B ‘→’ C) → Term {Γ} (A ‘→’ B
      _w‘‘’’ₐ_ : ∀ {Γ A B T} → Term {Γ ▷ T} (W (‘Term’ ‘’ ⌜ A ‘→’ B ⌝
      ‘‘’’ₐ’ : ∀ {Γ A B} → Term {Γ} (‘Term’ ‘’ ⌜ A ‘→’ B ⌝ ‘→’ ‘Term’ ‘’ ⌜
      - _w‘‘’’_ : ∀ {Γ A B T} → Term {Γ ▷ T} (‘Type’ (Γ ▷
      ‘‘□’’ : ∀ {Γ A B} → Term {Γ ▷ A ▷ B} (W (W (‘Term’ ‘’ ⌜ ‘Type’ Γ
      - ‘‘’’’ : ∀ {Γ A} → Term {Γ ▷ A} (‘Type’ (Γ ▷ A) ‘→
      _‘‘→’’_ : ∀ {Γ} → Term {Γ} (‘Type’ Γ) → Term {Γ} (‘Type’ Γ)
      _ww‘‘‘→’’’_ : ∀ {Γ A B} → Term {Γ ▷ A ▷ B} (W (W (‘Term’ ‘’ ⌜
      _ww‘‘‘×’’’_ : ∀ {Γ A B} → Term {Γ ▷ A ▷ B} (W (W (‘Term’ ‘’ ⌜

  □ : Type ε → Set _
  □ = Term {ε}

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

  _‘‘×’’_ : ∀ {Γ} → Term {Γ} (‘Type’ Γ) → Term {Γ} (‘Type’ Γ) → Te
  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
  ⟦_⟧ᵀ (W T) ⟦Γ⟧ = ⟦ T ⟧ᵀ (Σ.proj₁ ⟦Γ⟧)
  ⟦_⟧ᵀ (W₁ T) ⟦Γ⟧ = ⟦ T ⟧ᵀ ((Σ.proj₁ (Σ.proj₁ ⟦Γ⟧)) , (Σ.proj₂ ⟦Γ⟧))
  ⟦_⟧ᵀ (T ‘’ x) ⟦Γ⟧ = ⟦ T ⟧ᵀ (⟦Γ⟧ , ⟦ x ⟧ᵗ ⟦Γ⟧)
  ⟦_⟧ᵀ (‘Type’ Γ) ⟦Γ⟧ = Lifted (Type Γ)
  ⟦_⟧ᵀ ‘Term’ ⟦Γ⟧ = Lifted (Term (lower (Σ.proj₂ ⟦Γ⟧)))
  ⟦_⟧ᵀ (A ‘→’ B) ⟦Γ⟧ = ⟦ A ⟧ᵀ ⟦Γ⟧ → ⟦ B ⟧ᵀ ⟦Γ⟧
  ⟦_⟧ᵀ (A ‘×’ B) ⟦Γ⟧ = ⟦ A ⟧ᵀ ⟦Γ⟧ × ⟦ B ⟧ᵀ ⟦Γ⟧
  ⟦ ‘⊤’ ⟧ᵀ ⟦Γ⟧ = ⊤
  ⟦ ‘⊥’ ⟧ᵀ ⟦Γ⟧ = ⊥
  ⟦_⟧ᵀ (Quine φ) ⟦Γ⟧ = ⟦ φ ⟧ᵀ (⟦Γ⟧ , (lift (Quine φ)))

  ⟦_⟧ᵗ : ∀ {Γ : Context} {T : Type Γ} → Term T → (⟦Γ⟧ : ⟦ Γ ⟧ᶜ) → ⟦ T ⟧ᵀ ⟦Γ⟧
  ⟦_⟧ᵗ ⌜ x ⌝ ⟦Γ⟧ = lift x
  ⟦_⟧ᵗ ⌜ x ⌝ᵗ ⟦Γ⟧ = lift x
  ⟦_⟧ᵗ ‘⌜‘VAR₀’⌝ᵗ’ ⟦Γ⟧ = lift ⌜ (lower (Σ.proj₂ ⟦Γ⟧)) ⌝ᵗ
  ⟦_⟧ᵗ ‘⌜‘VAR₀’⌝’ ⟦Γ⟧ = lift ⌜ (lower (Σ.proj₂ ⟦Γ⟧)) ⌝
  ⟦_⟧ᵗ (f ‘’ₐ x) ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧ (⟦ x ⟧ᵗ ⟦Γ⟧)
  ⟦_⟧ᵗ ‘tt’ ⟦Γ⟧ = tt
  ⟦_⟧ᵗ (quine→ {φ}) ⟦Γ⟧ x = x
  ⟦_⟧ᵗ (quine← {φ}) ⟦Γ⟧ x = x
  ⟦_⟧ᵗ (‘λ•’ f) ⟦Γ⟧ x = ⟦ f ⟧ᵗ (⟦Γ⟧ , x)
  ⟦_⟧ᵗ ‘VAR₀’ ⟦Γ⟧ = Σ.proj₂ ⟦Γ⟧
  ⟦_⟧ᵗ (SW t) = ⟦_⟧ᵗ t
  ⟦_⟧ᵗ (←SW₁SV→W f) = ⟦ f ⟧ᵗ
  ⟦_⟧ᵗ (→SW₁SV→W f) = ⟦ f ⟧ᵗ
  ⟦_⟧ᵗ (←SW₁SV→SW₁SV→W f) = ⟦ f ⟧ᵗ
  ⟦_⟧ᵗ (→SW₁SV→SW₁SV→W f) = ⟦ f ⟧ᵗ
  ⟦_⟧ᵗ (w x) ⟦Γ⟧ = ⟦ x ⟧ᵗ (Σ.proj₁ ⟦Γ⟧)
  ⟦_⟧ᵗ (w→ f) ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦_⟧ᵗ (→w f) ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦_⟧ᵗ (ww→ f) ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦_⟧ᵗ (→ww f) ⟦Γ⟧ = ⟦ f ⟧ᵗ ⟦Γ⟧
  ⟦_⟧ᵗ ‘‘×’’ ⟦Γ⟧ A B = lift (lower A ‘×’ lower B)
  ⟦_⟧ᵗ (g ‘∘’ f) ⟦Γ⟧ x = ⟦ g ⟧ᵗ ⟦Γ⟧ (⟦ f ⟧ᵗ ⟦Γ⟧ x)
  ⟦_⟧ᵗ (f w‘‘’’ₐ x) ⟦Γ⟧ = lift (lower (⟦ f ⟧ᵗ ⟦Γ⟧) ‘’ₐ lower (⟦ x ⟧ᵗ ⟦Γ⟧))
  ⟦_⟧ᵗ ‘‘’ₐ’ ⟦Γ⟧ f x = lift (lower f ‘’ₐ lower x)
  ⟦_⟧ᵗ (‘‘□’’ {Γ} T) ⟦Γ⟧ = lift (‘Term’ ‘’ lower (⟦_⟧ᵗ T ⟦Γ⟧))
  ⟦_⟧ᵗ (A ‘‘→’’ B) ⟦Γ⟧ = lift ((lower (⟦_⟧ᵗ A ⟦Γ⟧)) ‘→’ (lower (⟦_⟧ᵗ B ⟦Γ⟧)))
  ⟦_⟧ᵗ (A ww‘‘‘→’’’ B) ⟦Γ⟧ = lift ((lower (⟦_⟧ᵗ A ⟦Γ⟧)) ‘‘→’’ (lower (⟦_⟧ᵗ B ⟦Γ⟧)))
  ⟦_⟧ᵗ (A ww‘‘‘×’’’ B) ⟦Γ⟧ = lift ((lower (⟦_⟧ᵗ A ⟦Γ⟧)) ‘‘×’’ (lower (⟦_⟧ᵗ B ⟦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} (
A w‘‘×’’ B = w→ (w→ (w ‘‘×’’’) ‘’ₐ A) ‘’ₐ B

löb : ∀ { ‘X’} → □ (‘□’ ‘X’ ‘→’ ‘X’) → ⟦ ‘X’ ⟧
löb f = ⟦_⟧ᵗ (Löb f) tt

¬_ : ∀ {ℓ} → Set ℓ → Set ℓ
¬_ {ℓ} T = T → ⊥ {ℓ}

incompleteness : ¬ □ (‘¬’ (‘□’ ‘⊥’))
incompleteness = löb

soundness : ¬ □ ‘⊥’
soundness x = ⟦ x ⟧ᵗ tt

non-emptyness : Σ (Type ε) (λ T → □ T)
non-emptyness = ‘⊤’ , ‘tt’
```

C. Encoding with Add-Quote Function

```
module lob-by-repr where
module well-typed-syntax where

infixl 2 _▻_
infixl 3 _‘’_
infixl 3 _‘’₁_
infixl 3 _‘’₂_
infixl 3 _‘’₃_
infixl 3 _‘’ₐ_
infixr 1 _‘→’_
infixl 3 _‘‘’’ₐ_
infixl 3 _w‘‘’’ₐ_
infixr 1 _‘‘→’’_
infixr 1 _w‘‘→’’_

mutual
  data Context : Set where
    ε : Context
    _▻_ : (Γ : Context) → Typ Γ → Context

  data Typ : Context → Set where
    _‘’_ : ∀ {Γ A} → Typ (Γ ▻ A) → Term {Γ} A → Typ Γ
    _‘’₁_ : ∀ {Γ A B} → (C : Typ (Γ ▻ A ▻ B)) → (a : Term {Γ} A) →
    _‘’₂_ : ∀ {Γ A B C} → (D : Typ (Γ ▻ A ▻ B ▻ C)) → (a : Term {Γ}
```

```
  _‘’₃_ : ∀ {Γ A B C D} → (E : Typ (Γ ▷ A ▷ B ▷ C ▷ D)) → (a : Term {Γ} A) → Typ (Γ ▷ B ‘’ a ▷ C ‘’₁ a ▷ D ‘’₂ a)
  W : ∀ {Γ A} → Typ Γ → Typ (Γ ▷ A)
  W1 : ∀ {Γ A B} → Typ (Γ ▷ B) → Typ (Γ ▷ A ▷ (W {Γ = Γ} {A = A} B))
  W2 : ∀ {Γ A B C} → Typ (Γ ▷ B ▷ C) → Typ (Γ ▷ A ▷ W B ▷ W1 C)
  _‘→’_ : ∀ {Γ} (A : Typ Γ) → Typ (Γ ▷ A) → Typ Γ
  ‘Σ’ : ∀ {Γ} (T : Typ Γ) → Typ (Γ ▷ T) → Typ Γ
  ‘Context’ : ∀ {Γ} → Typ Γ
  ‘Typ’ : ∀ {Γ} → Typ (Γ ▷ ‘Context’)
  ‘Term’ : ∀ {Γ} → Typ (Γ ▷ ‘Context’ ▷ ‘Typ’)

data Term : ∀ {Γ} → Typ Γ → Set where
  w : ∀ {Γ A B} → Term {Γ} B → Term {Γ = Γ ▷ A} (W {Γ = Γ} {A = A} B)
  ‘λ•’ : ∀ {Γ A B} → Term {Γ = (Γ ▷ A)} B → Term {Γ} (A ‘→’ B)
  _‘’ₐ_ : ∀ {Γ A B} → (f : Term {Γ} (A ‘→’ B)) → (x : Term {Γ} A) → Term {Γ} (B ‘’ x)
  ‘VAR₀’ : ∀ {Γ T} → Term {Γ = Γ ▷ T} (W T)
  ⌜_⌝c : ∀ {Γ} → Context → Term {Γ} ‘Context’
  ⌜_⌝T : ∀ {Γ Γ’} → Typ Γ’ → Term {Γ} (‘Typ’ ‘’ ⌜ Γ’ ⌝c)
  ⌜_⌝t : ∀ {Γ Γ’} {T : Typ Γ’} → Term T → Term {Γ} (‘Term’ ‘’₁ ⌜ Γ’ ⌝c ‘’ ⌜ T ⌝T)
  ‘quote-term’ : ∀ {Γ Γ’} {A : Typ Γ’} → Term {Γ} (‘Term’ ‘’₁ ⌜ Γ’ ⌝c ‘’ ⌜ A ⌝T
  ‘quote-sigma’ : ∀ {Γ Γ’} → Term {Γ} (‘Σ’ ‘Context’ ‘Typ’ ‘→’ W (‘Term’ ‘’₁ ⌜ Γ’ ⌝c ‘’ ⌜ ‘Σ’ ‘Context’ ‘Typ’ ⌝T))
  ‘cast’ : Term {ε} (‘Σ’ ‘Context’ ‘Typ’ ‘→’ W (‘Typ’ ‘’ ⌜ ε ▷ ‘Σ’ ‘Context’ ‘Typ’ ⌝c))
  SW : ∀ {Γ A B} {a : Term {Γ} A} → Term {Γ} (W B ‘’ a) → Term {Γ} B
  weakenTyp-substTyp-tProd : ∀ {Γ T T’ A B} {a : Term {Γ} T} → Term {Γ = Γ ▷ T’ ‘’ a}
  substTyp-weakenTyp1-VAR₀ : ∀ {Γ A T} → Term {Γ ▷ A} (W1 T ‘’ ‘VAR₀’)
  weakenTyp-tProd : ∀ {Γ A B C} → Term {Γ = Γ ▷ C} (W (A ‘→’ B)) → Term {Γ = Γ ▷ C} (W A ‘→’ W1 B)
  weakenTyp-tProd-inv : ∀ {Γ A B C} → Term {Γ = Γ ▷ C} (W A ‘→’ W1 B) → Term {Γ = Γ ▷ C} (W (A ‘→’ B))
  weakenTyp-weakenTyp-tProd : ∀ {Γ A B C D} → Term {Γ ▷ C ▷ D} (W (W (A ‘→’ B))) → Term {Γ ▷ C ▷ D} (W (W A ‘→’ W1 B))
  substTyp1-tProd : ∀ {Γ T T’ A B} {a : Term {Γ} T} → Term {Γ ▷ T’ ‘’ a} ((A ‘→’ B) ‘’₁ a) → Term {Γ ▷ T’ ‘’ a} ((A ‘’₁ a) ‘→’ (B ‘’₂ a))
  weakenTyp1-tProd : ∀ {Γ C D A B} → Term {Γ ▷ C ▷ W D} (W1 (A ‘→’ B)) → Term {Γ ▷ C ▷ W D} (W1 A ‘→’ W2 B)
  substTyp2-tProd : ∀ {Γ T T’ T” A B} {a : Term {Γ} T} → Term {Γ ▷ T’ ‘’ a ▷ T” ‘’₁ a} ((A ‘→’ B) ‘’₂ a) → Term {Γ ▷ T’ ‘’ a ▷ T” ‘’₁ a}
  substTyp1-substTyp-weakenTyp-inv : ∀ {Γ C T A} {a : Term {Γ} C} {b : Term {Γ} (T ‘’ a)} → Term {Γ} (A ‘’ a) → Term {Γ} (W A ‘’₁ a ‘’ b)
  substTyp1-substTyp-weakenTyp : ∀ {Γ C T A} {a : Term {Γ} C} {b : Term {Γ} (T ‘’ a)} → Term {Γ} (W A ‘’₁ a ‘’ b) → Term {Γ} (A ‘’ a)
  weakenTyp-weakenTyp-substTyp1-substTyp-weakenTyp : ∀ {Γ C T A D E} {a : Term {Γ} C} {b : Term {Γ} (T ‘’ a)} → Term {Γ ▷ D ▷ E
  weakenTyp-substTyp2-substTyp1-substTyp-weakenTyp-inv : ∀ {Γ A B C T T’} {a : Term {Γ} A} {b : Term {Γ} (B ‘’ a)} {c : Term {Γ} (C
    → Term {Γ ▷ T’} (W (T ‘’₁ a ‘’ b))
    → Term {Γ ▷ T’} (W (W T ‘’₂ a ‘’₁ b ‘’ c))
  substTyp2-substTyp1-substTyp-weakenTyp : ∀ {Γ A B C T} {a : Term {Γ} A} {b : Term {Γ} (B ‘’ a)} {c : Term {Γ} (C ‘’₁ a ‘’ b)}
    → Term {Γ} (W T ‘’₂ a ‘’₁ b ‘’ c)
    → Term {Γ} (T ‘’₁ a ‘’ b)
  weakenTyp-substTyp2-substTyp1-substTyp-tProd : ∀ {Γ T T’ T” T”’ A B} {a : Term {Γ} T} {b : Term {Γ} (T’ ‘’ a)} {c : Term {Γ} (T” ‘
    → Term {Γ ▷ T”’} (W ((A ‘→’ B) ‘’₂ a ‘’₁ b ‘’ c))
    → Term {Γ ▷ T”’} ((W (A ‘’₂ a ‘’₁ b ‘’ c)) ‘→’ (W1 (B ‘’₃ a ‘’₂ b ‘’₁ c)))
  weakenTyp2-weakenTyp1 : ∀ {Γ A B C D} → Term {Γ ▷ A ▷ W B ▷ W1 C} (W2 (W D)) → Term {Γ ▷ A ▷ W B ▷ W1 C} (W (W1 D))
  weakenTyp1-weakenTyp : ∀ {Γ A B C} → Term {Γ ▷ A ▷ W B} (W1 (W C)) → Term {Γ ▷ A ▷ W B} (W (W C))
  weakenTyp1-weakenTyp-inv : ∀ {Γ A B C} → Term {Γ ▷ A ▷ W B} (W (W C)) → Term {Γ ▷ A ▷ W B} (W1 (W C))
  weakenTyp1-weakenTyp1-weakenTyp : ∀ {Γ A B C T} → Term {Γ ▷ A ▷ B ▷ W (W C)} (W1 (W1 (W T))) → Term {Γ ▷ A ▷ B ▷ W (W C
  substTyp1-weakenTyp1 : ∀ {Γ A B C} {a : Term {Γ} A} → Term {Γ ▷ W B ‘’ a} (W1 C ‘’₁ a) → Term {Γ ▷ B} C
  weakenTyp1-substTyp-weakenTyp1-inv : ∀ {Γ A T” T’ T} {a : Term {Γ} A}
    → Term {Γ ▷ T” ▷ W (T’ ‘’ a)} (W1 (W (T ‘’ a)))
    → Term {Γ ▷ T” ▷ W (T’ ‘’ a)} (W1 (W T ‘’₁ a))
  weakenTyp1-substTyp-weakenTyp1 : ∀ {Γ A T” T’ T} {a : Term {Γ} A}
    → Term {Γ ▷ T” ▷ W (T’ ‘’ a)} (W1 (W T ‘’₁ a))
    → Term {Γ ▷ T” ▷ W (T’ ‘’ a)} (W1 (W (T ‘’ a)))
  weakenTyp-substTyp-substTyp-weakenTyp1 : ∀ {Γ T’ B A} {b : Term {Γ} B} {a : Term {Γ ▷ B} (W A)} {T : Typ (Γ ▷ A)}
    → Term {Γ ▷ T’} (W (W1 T ‘’ a ‘’ b))
    → Term {Γ ▷ T’} (W (T ‘’ (SW ((‘λ•’ a) ‘’ₐ b))))
  weakenTyp-substTyp-substTyp-weakenTyp1-inv : ∀ {Γ T’ B A} {b : Term {Γ} B} {a : Term {Γ ▷ B} (W A)} {T : Typ (Γ ▷ A)}
    → Term {Γ ▷ T’} (W (T ‘’ (SW ((‘λ•’ a) ‘’ₐ b))))
    → Term {Γ ▷ T’} (W (W1 T ‘’ a ‘’ b))
  substTyp-weakenTyp1-weakenTyp : ∀ {Γ T} {A : Typ Γ} {B : Typ Γ}
    → {a : Term {Γ = Γ ▷ T} (W {Γ = Γ} {A = T} B)}
```

```
    → Term {Γ = Γ ▷ T} (W A)
  substTyp3-substTyp2-substTyp1-substTyp-weakenTyp : ∀ {Γ A B C
    {d : Term {Γ = (Γ ▷ T’)} (W (D ‘’₂ a ‘’₁ b ‘’ c))}
    → Term {Γ = (Γ ▷ T’)} (W1 (W T ‘’₃ a ‘’₂ b ‘’₁ c) ‘’ d)
    → Term {Γ = (Γ ▷ T’)} (W (T ‘’₂ a ‘’₁ b ‘’ c))
  weakenTyp-substTyp2-substTyp1-substTyp-weakenTyp1 : ∀ {Γ A B
    → Term {Γ = (Γ ▷ T’)} (W (W1 T ‘’₂ a ‘’₁ b ‘’ substTyp1-subst
    → Term {Γ = (Γ ▷ T’)} (W (T ‘’₁ a ‘’ c))
  substTyp1-substTyp-tProd : ∀ {Γ T T’ A B a b} → Term ((_‘→’_
  substTyp2-substTyp-substTyp-weakenTyp1-weakenTyp-weakenTy
    {c : Term {Γ = (Γ ▷ T’)} (W (C ‘’ a))}
    → Term {Γ = (Γ ▷ T’)} (W1 (W (W T) ‘’₂ a ‘’ b) ‘’ c)
    → Term {Γ = (Γ ▷ T’)} (W (T ‘’ a))
  substTyp1-substTyp-weakenTyp2-weakenTyp : ∀ {Γ T’ A B T} {a :
    → Term {Γ ▷ T’} (W2 (W T) ‘’₁ a ‘’ b)
    → Term {Γ ▷ T’} (W1 T ‘’ a)
  weakenTyp-weakenTyp1-weakenTyp : ∀ {Γ A B C D} → Term {Γ ▷
  beta-under-subst : ∀ {Γ A B B’} {g : Term {Γ} (A ‘→’ W B)} {x : T
  ‘proj₁’ : ∀ {Γ} {T : Typ Γ} {P : Typ (Γ ▷ T)} → Term (‘Σ’ T P ‘→
  ‘proj₂’’ : ∀ {Γ} {T : Typ Γ} {P : Typ (Γ ▷ T)} → Term {Γ ▷ ‘Σ’ T P
  ‘existT’ : ∀ {Γ T P} (x : Term T) (p : Term (P ‘’ x)) → Term
  {- these are redundant, but useful for not having to r
    → Term {ε} (‘Typ’ ‘’ Γ)
    → Term {ε} (‘Typ’ ‘’ Γ)
  _w‘‘’’_ : ∀ {X Γ} {b : Term {Γ} (B ‘’ a)} {c : Term {Γ} (C ‘’₁ a ‘’ b)}
    → Term {ε ▷ X} (W (‘Typ’ ‘’ Γ))
    → Term {ε ▷ X} (W (‘Typ’ ‘’ Γ))
  w→ : ∀ {Γ A B C} → Term (A ‘→’ W B) → Term {Γ = Γ ▷ C} (W
  {- things that were postulates, but are no longer -}
  w‘‘→’’’→‘‘→’’’ : ∀ {T’}
    {b : Term {ε} (‘Typ’ ‘’ ⌜ ε ⌝c)}
    {c : Term {ε ▷ T’} (W (‘Typ’ ‘’ ⌜ ε ⌝c))}
    {e : Term {ε} T’}
    → Term {ε} (‘Term’ ‘’₁ ⌜ ε ⌝c ‘’ (SW (‘λ•’ c ‘’ₐ e) ‘‘→’’’ b)
    → Term {ε} (‘Term’ ‘’₁ ⌜ ε ⌝c ‘’ (SW (‘λ•’ (c w‘‘→’’’ w b) ‘’ₐ e))))
  ‘tApp-nd’ : ∀ {Γ} {A : Term {ε} (‘Typ’ ‘’ Γ)} {B : Term {ε} (‘Typ’
    Term {ε} (‘Term’ ‘’₁ Γ ‘’ (A ‘‘→’’’ B)
      ‘→’ W (‘Term’ ‘’₁ Γ ‘’ B)))
  ⌜←’⌝ : ∀ {H X} →
    Term {ε} (‘Term’ ‘’₁ ⌜ ε ⌝c ‘’ (⌜ H ⌝T ‘‘→’’’ ⌜ X ⌝T)
      ‘→’ W (‘Term’ ‘’₁ ⌜ ε ⌝c ‘’ ⌜ H ‘→’ W X ⌝T))
```

```
⌜→'⌝ : ∀ {H X} →
  Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c ‘’ ⌜ H ‘→’ W X ⌝T
    ‘→’ W (‘Term’ ‘’1 ⌜ ε ⌝c ‘’ (⌜ H ⌝T ‘’→‘’ ⌜ X ⌝T)))
‘‘fcomp-nd’’ : ∀ {A B C} →
  Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c ‘’ (A ‘’→‘’ C)
    ‘→’ W (‘Term’ ‘’1 ⌜ ε ⌝c ‘’ (C ‘’→‘’ B)
    ‘→’ W (‘Term’ ‘’1 ⌜ ε ⌝c ‘’ (A ‘’→‘’ B))))
⌜‘’⌝ : ∀ {B A} {b : Term {ε} B} →
  Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c ‘’
    (⌜ A ‘’ b ⌝T ‘’→‘’ ⌜ A ⌝T ‘’‘’ ⌜ b ⌝t))
⌜‘’⌝' : ∀ {B A} {b : Term {ε} B} →
  Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c ‘’
    (⌜ A ⌝T ‘’‘’ ⌜ b ⌝t ‘’→‘’ ⌜ A ‘’ b ⌝T))
‘cast-refl’ : ∀ {T : Typ (ε ▷ ‘Σ’ ‘Context’ ‘Typ’)} →
  Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c ‘’
    ((⌜ T ‘’ ‘existT’ ⌜ ε ▷ ‘Σ’ ‘Context’ ‘Typ’ ⌝c ⌜ T ⌝T ⌝T)
      ‘’→‘’
      (SW (‘cast’ ‘’a ‘existT’ ⌜ ε ▷ ‘Σ’ ‘Context’ ‘Typ’ ⌝c ⌜ T ⌝T)
        ‘’‘’ SW (‘quote-sigma’ ‘’a ‘existT’ ⌜ ε ▷ ‘Σ’ ‘Context’ ‘Typ’ ⌝c ⌜ T ⌝T))))
‘cast-refl’’ : ∀ {T : Typ (ε ▷ ‘Σ’ ‘Context’ ‘Typ’)} →
  Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c ‘’
    ((SW (‘cast’ ‘’a ‘existT’ ⌜ ε ▷ ‘Σ’ ‘Context’ ‘Typ’ ⌝c ⌜ T ⌝T)
        ‘’‘’ SW (‘quote-sigma’ ‘’a ‘existT’ ⌜ ε ▷ ‘Σ’ ‘Context’ ‘Typ’ ⌝c ⌜ T ⌝T))
      ‘’→‘’
      (⌜ T ‘’ ‘existT’ ⌜ ε ▷ ‘Σ’ ‘Context’ ‘Typ’ ⌝c ⌜ T ⌝T ⌝T)))
‘s→→’ : ∀ {T B}
    {b : Term {ε} (T ‘→’ W (‘Typ’ ‘’ ⌜ ε ▷ B ⌝c))}
    {c : Term {ε} (T ‘→’ W (‘Term’ ‘’1 ⌜ ε ⌝c ‘’ ⌜ B ⌝T))}
    {v : Term {ε} T} →
  (Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c
    ‘’ ((SW (((‘λ•’ (SW (w→ b ‘’a ‘VAR0’) w‘’‘’ SW (w→ c ‘’a ‘VAR0’)) ‘’a v))
      ‘’→‘’ (SW (b ‘’a v) ‘’‘’ SW (c ‘’a v)))))
‘s←←’ : ∀ {T B}
    {b : Term {ε} (T ‘→’ W (‘Typ’ ‘’ ⌜ ε ▷ B ⌝c))}
    {c : Term {ε} (T ‘→’ W (‘Term’ ‘’1 ⌜ ε ⌝c ‘’ ⌜ B ⌝T))}
    {v : Term {ε} T} →
  (Term {ε} (‘Term’ ‘’1 ⌜ ε ⌝c
    ‘’ ((SW (b ‘’a v) ‘’‘’ SW (c ‘’a v))
      ‘’→‘’ (SW (((‘λ•’ (SW (w→ b ‘’a ‘VAR0’) w‘’‘’ SW (w→ c ‘’a ‘VAR0’)) ‘’a v)))))))

module well-typed-syntax-helpers where
  open well-typed-syntax

  infixl 3 _‘’ₐ_
  infixr 1 _‘→’’_
  infixl 3 _‘t’_
  infixl 3 _‘t’1_
  infixl 3 _‘t’2_
  infixr 2 _‘∘’_

  WS∀ : ∀ {Γ T T’ A B} {a : Term {Γ = Γ} T} → Term {Γ = Γ ▷ T’} (W ((A ‘→’ B) ‘’ a)) → Term {Γ = Γ ▷ T’} (W (A ‘→’ B) ‘’ a)
  WS∀ = weakenTyp-substTyp-tProd

  _‘→’’_ : ∀ {Γ} → Typ Γ → Typ Γ → Typ Γ
  _‘→’’_ {Γ = Γ} A B = _‘→’_ {Γ = Γ} A (W {Γ = Γ} {A = A} B)

  _‘’ₐ_ : ∀ {Γ A B} → Term {Γ} (A ‘→’’ B) → Term A → Term B
  _‘’ₐ_ {Γ} {A} {B} f x = SW (_‘’a_ {Γ} {A} {W B} f x)

  _‘t’_ : ∀ {Γ A} {B : Typ (Γ ▷ A)} → (b : Term {Γ = Γ ▷ A} B) → (a : Term {Γ} A) → Term {Γ} (B ‘’ a)
  b ‘t’ a = ‘λ•’ b ‘’a a
```

```
substTyp-tProd : ∀ {Γ T A B} {a : Term {Γ} T} →
  Term {Γ} ((A ‘→’ B) ‘’ a)
  → Term {Γ} (_‘→’_ {Γ = Γ} (A ‘’ a) (B ‘’1 a))
substTyp-tProd {Γ} {T} {A} {B} {a} x = SW ((WS∀ (w x)) ‘t’ a)

S∀ = substTyp-tProd

‘λ’• : ∀ {Γ A B} → Term {Γ ▷ A} (W B) -> Term (A ‘→’’ B)
‘λ’• f = ‘λ•’ f

SW1V : ∀ {Γ A T} → Term {Γ ▷ A} (W1 T ‘’ ‘VAR0’) → Term {Γ ▷ A} T
SW1V = substTyp-weakenTyp1-VAR0

S1∀ : ∀ {Γ T T’ A B} {a : Term {Γ} T} → Term {Γ ▷ T’ ‘’ a} ((A ‘→’ B)
S1∀ = substTyp1-tProd

un‘λ•’ : ∀ {Γ A B} → Term (A ‘→’ B) → Term {Γ ▷ A} B
un‘λ•’ f = SW1V (weakenTyp-tProd (w f) ‘’a ‘VAR0’)

weakenProd : ∀ {Γ A B C} →
  Term {Γ} (A ‘→’ B)
  → Term {Γ = Γ ▷ C} (W A ‘→’ W1 B)
weakenProd {Γ} {A} {B} {C} x = weakenTyp-tProd (w x)
w∀ = weakenProd

w1 : ∀ {Γ A B C} → Term {Γ = Γ ▷ B} C → Term {Γ = Γ ▷ A ▷ W {Γ =
w1 x = un‘λ•’ (weakenTyp-tProd (w (‘λ•’ x)))

_‘t’1_ : ∀ {Γ A B C} → (c : Term {Γ = Γ ▷ A ▷ B} C) → (a : Term {Γ}
f ‘t’1 x = un‘λ•’ (S∀ (‘λ•’ (‘λ•’ f) ‘’a x))
_‘t’2_ : ∀ {Γ A B C D} → (c : Term {Γ = Γ ▷ A ▷ B ▷ C} D) → (a : Ter
f ‘t’2 x = un‘λ•’ (S1∀ (un‘λ•’ (S∀ (‘λ•’ (‘λ•’ (‘λ•’ f)) ‘’a x))))

S10W’ : ∀ {Γ C T A} {a : Term {Γ} C} {b : Term {Γ} (T ‘’ a)} → Term
S10W’ = substTyp1-substTyp-weakenTyp-inv

S10W : ∀ {Γ C T A} {a : Term {Γ} C} {b : Term {Γ} (T ‘’ a)} → Term {
S10W = substTyp1-substTyp-weakenTyp

substTyp1-substTyp-weakenTyp-weakenTyp : ∀ {Γ T A} {B : Typ (Γ ▷ A
  → {a : Term {Γ} A}
  → {b : Term {Γ} (B ‘’ a)}
  → Term {Γ} (W (W T) ‘’1 a ‘’ b)
  → Term {Γ} T
substTyp1-substTyp-weakenTyp-weakenTyp x = SW (S10W x)

S10WW = substTyp1-substTyp-weakenTyp-weakenTyp


S210W : ∀ {Γ A B C T} {a : Term {Γ} A} {b : Term {Γ} (B ‘’ a)} {c : Te
  → Term {Γ} (T ‘’1 a ‘’ b)
S210W = substTyp2-substTyp1-substTyp-weakenTyp

substTyp2-substTyp1-substTyp-weakenTyp : ∀ {Γ A B C T}
  {a : Term {Γ} A}
  {b : Term {Γ} (B ‘’ a)}
  {c : Term {Γ} (C ‘’1 a ‘’ b)} →
  Term {Γ} (W (W T) ‘’2 a ‘’1 b ‘’ c)
  → Term {Γ} (B ‘’ a)
substTyp2-substTyp1-substTyp-weakenTyp-weakenTyp x = S10W (S210
```

```
S210WW = substTyp2-substTyp1-substTyp-weakenTyp-weakenTyp

W1W : ∀ {Γ A B C} → Term {Γ ▻ A ▻ W B} (W1 (W C)) → Term {Γ ▻ A ▻ W B} (W (W C))
W1W = weakenTyp1-weakenTyp

W1W1W : ∀ {Γ A B C T} → Term {Γ ▻ A ▻ B ▻ W (W C)} (W1 (W1 (W1 T))) → Term {Γ ▻ A ▻ B ▻ W (W C)} (W1 (W1 (W T)))
W1W1W = weakenTyp1-weakenTyp1-weakenTyp

weakenTyp-tProd-nd : ∀ {Γ A B C} →
  Term {Γ = Γ ▻ C} (W (A ‘→’' B))
  → Term {Γ = Γ ▻ C} (W A ‘→’' W B)
weakenTyp-tProd-nd x = ‘λ’• (W1W (SW1V (weakenTyp-tProd (w (weakenTyp-tProd (w1 x)) ‘’ₐ ‘VAR0’))))

weakenProd-nd : ∀ {Γ A B C} →
  Term (A ‘→’' B)
  → Term {Γ = Γ ▻ C} (W A ‘→’' W B)
weakenProd-nd {Γ} {A} {B} {C} x = weakenTyp-tProd-nd (w x)




weakenTyp-tProd-nd-tProd-nd : ∀ {Γ A B C D} →
  Term {Γ = Γ ▻ D} (W (A ‘→’' B ‘→’' C))
  → Term {Γ = Γ ▻ D} (W A ‘→’' W B ‘→’' W C)
weakenTyp-tProd-nd-tProd-nd x = ‘λ•’ (weakenTyp-tProd-inv (‘λ•’ (W1W1W (SW1V (w∀ (weakenTyp-tProd (weakenTyp-weakenTyp-tPro

weakenProd-nd-Prod-nd : ∀ {Γ A B C D} →
  Term (A ‘→’' B ‘→’' C)
  → Term {Γ = Γ ▻ D} (W A ‘→’' W B ‘→’' W C)
weakenProd-nd-Prod-nd {Γ} {A} {B} {C} {D} x = weakenTyp-tProd-nd-tProd-nd (w x)
w→→ = weakenProd-nd-Prod-nd

S1W1 : ∀ {Γ A B C} {a : Term {Γ} A} → Term {Γ ▻ W B '' a} (W1 C ''1 a) → Term {Γ ▻ B} C
S1W1 = substTyp1-weakenTyp1


W1S1W' : ∀ {Γ A T'' T' T} {a : Term {Γ} A}
  → Term {Γ ▻ T'' ▻ W (T' '' a)} (W1 (W (T '' a)))
  → Term {Γ ▻ T'' ▻ W (T' '' a)} (W1 (W T ''1 a))
W1S1W' = weakenTyp1-substTyp-weakenTyp1-inv


substTyp-weakenTyp1-inv : ∀ {Γ A T' T}
  {a : Term {Γ} A} →
  Term {Γ = (Γ ▻ T' '' a)} (W (T '' a))
  → Term {Γ = (Γ ▻ T' '' a)} (W T ''1 a)
substTyp-weakenTyp1-inv {a = a} x = S1W1 (W1S1W' (w1 x) ‘t’1 a)

S1W' = substTyp-weakenTyp1-inv

_‘∘’_ : ∀ {Γ A B C}
  → Term {Γ} (A ‘→’' B)
  → Term {Γ} (B ‘→’' C)
  → Term {Γ} (A ‘→’' C)
g ‘∘’ f = ‘λ•’ (w→ f ‘’'ₐ (w→ g ‘’'ₐ ‘VAR0’))


WS00W1 : ∀ {Γ T' B A} {b : Term {Γ} B} {a : Term {Γ ▻ B} (W A)} {T : Typ (Γ ▻ A)}
  → Term {Γ ▻ T'} (W (W1 T '' a '' b))
  → Term {Γ ▻ T'} (W (T '' (SW (a ‘t’ b))))
WS00W1 = weakenTyp-substTyp-substTyp-weakenTyp1
```

```
WS00W1' : ∀ {Γ T' B A} {b : Term {Γ} B} {a : Term {Γ ▻ B} (W A)} {T
  → Term {Γ ▻ T'} (W (T '' (SW (a ‘t’ b))))
  → Term {Γ ▻ T'} (W (W1 T '' a '' b))
WS00W1' = weakenTyp-substTyp-substTyp-weakenTyp1-inv

substTyp-substTyp-weakenTyp1-inv-arr : ∀ {Γ B A}
  {b : Term {Γ} B}
  {a : Term {Γ ▻ B} (W A)}
  {T : Typ (Γ ▻ A)}
  {X} →
  Term {Γ} (T '' (SW (a ‘t’ b)) ‘→’' X)
  → Term {Γ} (W1 T '' a '' b ‘→’' X)
substTyp-substTyp-weakenTyp1-inv-arr x = ‘λ•’ (w→ x ‘’'ₐ WS00W1 ‘V

S00W1'→ = substTyp-substTyp-weakenTyp1-inv-arr

substTyp-substTyp-weakenTyp1-arr-inv : ∀ {Γ B A}
  {b : Term {Γ} B}
  {a : Term {Γ ▻ B} (W A)}
  {T : Typ (Γ ▻ A)}
  {X} →
  Term {Γ} (X ‘→’' T '' (SW (a ‘t’ b)))
  → Term {Γ} (X ‘→’' W1 T '' a '' b)
substTyp-substTyp-weakenTyp1-arr-inv x = ‘λ•’ (WS00W1' (un‘λ•’ x))
S00W1'← = substTyp-substTyp-weakenTyp1-arr-inv


substTyp-substTyp-weakenTyp1 : ∀ {Γ B A}
  {b : Term {Γ} B}
  {a : Term {Γ ▻ B} (W A)}
  {T : Typ (Γ ▻ A)} →
  Term {Γ} (W1 T '' a '' b)
  → Term {Γ} (T '' (SW (a ‘t’ b)))
substTyp-substTyp-weakenTyp1 x = (SW (WS00W1 (w x) ‘t’ x))
S00W1 = substTyp-substTyp-weakenTyp1

SW1W : ∀ {Γ T} {A : Typ Γ} {B : Typ Γ}
  → {a : Term {Γ = Γ ▻ T} (W {Γ = Γ} {A = T} B)}
  → Term {Γ = Γ ▻ T} (W1 (W A) '' a)
  → Term {Γ = Γ ▻ T} (W A)
SW1W = substTyp-weakenTyp1-weakenTyp


S200W1WW : ∀ {Γ A} {T : Typ (Γ ▻ A)} {T' C B} {a : Term {Γ} A} {b
  {c : Term {Γ = (Γ ▻ T')} (W (C '' a))}
  → Term {Γ = (Γ ▻ T')} (W1 (W (W T) ''2 a '' b) '' c)
  → Term {Γ = (Γ ▻ T')} (W (T '' a))
S200W1WW = substTyp2-substTyp-substTyp-weakenTyp1-weakenTyp-


S10W2W : ∀ {Γ T' A B T} {a : Term {Γ ▻ T'} (W A)} {b : Term {Γ ▻ T'
  → Term {Γ ▻ T'} (W2 (W T) ''1 a '' b)
  → Term {Γ ▻ T'} (W1 T '' a)
S10W2W = substTyp1-substTyp-weakenTyp2-weakenTyp
module well-typed-syntax-context-helpers where
  open well-typed-syntax
  open well-typed-syntax-helpers

  □_ : Typ ε → Set
  □_ T = Term {Γ = ε} T
```

```
module well-typed-quoted-syntax-defs where
  open well-typed-syntax
  open well-typed-syntax-helpers
  open well-typed-syntax-context-helpers

  ‘ε’ : Term {Γ = ε} ‘Context’
  ‘ε’ = ⌜ ε ⌝c

  ‘□’ : Typ (ε ▻ ‘Typ’ ‘’ ‘ε’)
  ‘□’ = ‘Term’ ‘’_1 ‘ε’

module well-typed-syntax-eq-dec where
  open well-typed-syntax

  context-pick-if : ∀ {ℓ} {P : Context → Set ℓ}
    {Γ : Context}
    (dummy : P (ε ▻ ‘Σ’ ‘Context’ ‘Typ’))
    (val : P Γ) →
  P (ε ▻ ‘Σ’ ‘Context’ ‘Typ’)
  context-pick-if {P = P} {ε ▻ ‘Σ’ ‘Context’ ‘Typ’} dummy val = val
  context-pick-if {P = P} {Γ} dummy val = dummy

  context-pick-if-refl : ∀ {ℓ P dummy val} →
    context-pick-if {ℓ} {P} {ε ▻ ‘Σ’ ‘Context’ ‘Typ’} dummy val ≡ val
  context-pick-if-refl {P = P} = refl

module well-typed-quoted-syntax where
  open well-typed-syntax
  open well-typed-syntax-helpers public
  open well-typed-quoted-syntax-defs public
  open well-typed-syntax-context-helpers public
  open well-typed-syntax-eq-dec public

  infixr 2 _‘‘∘’’_

  quote-sigma : (Γv : Σ Context Typ) → Term {ε} (‘Σ’ ‘Context’ ‘Typ’)
  quote-sigma (Γ , v) = ‘existT’ ⌜ Γ ⌝c ⌜ v ⌝T

  _‘‘∘’’_ : ∀ {A B C}
    → □ (‘□’ ‘’ (C ‘‘→’’ B))
    → □ (‘□’ ‘’ (A ‘‘→’’ C))
    → □ (‘□’ ‘’ (A ‘‘→’’ B))
  g ‘‘∘’’ f = (‘‘fcomp-nd’’ ‘’ₐ f ‘’ₐ g)

  Conv0 : ∀ {qH0 qX} →
    Term {Γ = (ε ▻ ‘□’ ‘’ qH0)}
      (W (‘□’ ‘’ ⌜ ‘□’ ‘’ qH0 ‘→’’ qX ⌝T))
    → Term {Γ = (ε ▻ ‘□’ ‘’ qH0)}
      (W
        (‘□’ ‘’ (⌜ ‘□’ ‘’ qH0 ⌝T ‘‘→’’ ⌜ qX ⌝T)))
  Conv0 {qH0} {qX} x = w→ ⌜→’⌝ ‘’ₐ x

module well-typed-syntax-pre-interpreter where
  open well-typed-syntax
  open well-typed-syntax-helpers

  max-level : Level
  max-level = lsuc lzero

  module inner
    (context-pick-if’ : ∀ ℓ (P : Context → Set ℓ)
      (Γ : Context)
      (dummy : P (ε ▻ ‘Σ’ ‘Context’ ‘Typ’))
      (val : P Γ) →
    P (ε ▻ ‘Σ’ ‘Context’ ‘Typ’))
    (context-pick-if-refl’ : ∀ ℓ P dummy val →
      context-pick-if’ ℓ P (ε ▻ ‘Σ’ ‘Context’ ‘Typ’) dummy val ≡ val)
    where

    context-pick-if : ∀ {ℓ} {P : Context → Set ℓ}
      {Γ : Context}
      (dummy : P (ε ▻ ‘Σ’ ‘Context’ ‘Typ’))
      (val : P Γ) →
    P (ε ▻ ‘Σ’ ‘Context’ ‘Typ’)
    context-pick-if {P = P} dummy val = context-pick-if’ _ P _ dummy va
    context-pick-if-refl : ∀ {ℓ P dummy val} →
      context-pick-if {ℓ} {P} {ε ▻ ‘Σ’ ‘Context’ ‘Typ’} dummy val ≡ val
    context-pick-if-refl {P = P} = context-pick-if-refl’ _ P _ _

    private
      dummy : Typ ε
      dummy = ‘Context’

    cast-helper : ∀ {X T A} {x : Term X} → A ≡ T → Term {ε} (T ‘’ x ‘→
    cast-helper refl = ‘λ∙’ ‘VAR_0’

    cast’-proof : ∀ {T} → Term {ε} (context-pick-if {P = Typ} (W dumm
      ‘→’’ T ‘’ ‘existT’ ⌜ ε ▻ ‘Σ’ ‘Context’ ‘Typ’ ⌝c ⌜ T ⌝T)
    cast’-proof {T} = cast-helper {‘Σ’ ‘Context’ ‘Typ’}
      {context-pick-if {P = Typ} {ε ▻ ‘Σ’ ‘Context’ ‘Typ’} (W dummy)
      {T} (sym (context-pick-if-refl {P = Typ} {dummy = W dummy}))

    cast-proof : ∀ {T} → Term {ε} (T ‘’ ‘existT’ ⌜ ε ▻ ‘Σ’ ‘Context’ ‘Typ’
      ‘→’’ context-pick-if {P = Typ} (W dummy) T ‘’ ‘existT’ ⌜ ε ▻ ‘Σ’ ‘
    cast-proof {T} = cast-helper {‘Σ’ ‘Context’ ‘Typ’} {T}
      {context-pick-if {P = Typ} {ε ▻ ‘Σ’ ‘Context’ ‘Typ’} (W dummy)
      (context-pick-if-refl {P = Typ} {dummy = W dummy})

    ‘idfun’ : ∀ {T} → Term {ε} (T ‘→’’ T)
    ‘idfun’ = ‘λ∙’ ‘VAR_0’

    mutual
      Context⇓ : (Γ : Context) → Set (lsuc max-level)
      Typ⇓ : {Γ : Context} → Typ Γ → Context⇓ Γ → Set max-level

      Context⇓ ε = ⊤
      Context⇓ (Γ ▻ T) = Σ (Context⇓ Γ) (λ Γ’ → Typ⇓ T Γ’)

      Typ⇓ (T_1 ‘’ x) Γ⇓ = Typ⇓ T_1 (Γ⇓ , Term⇓ x Γ⇓)
      Typ⇓ (T_2 ‘’_1 a) (Γ⇓ , A⇓) = Typ⇓ T_2 ((Γ⇓ , Term⇓ a Γ⇓) , A⇓)
      Typ⇓ (T_3 ‘’_2 a) ((Γ⇓ , A⇓) , B⇓) = Typ⇓ T_3 (((Γ⇓ , Term⇓ a Γ⇓) ,
      Typ⇓ (T_3 ‘’_3 a) (((Γ⇓ , A⇓) , B⇓) , C⇓) = Typ⇓ T_3 ((((Γ⇓ , Term⇓
      Typ⇓ (W T_1) (Γ⇓ , _) = Typ⇓ T_1 Γ⇓
      Typ⇓ (W1 T_2) ((Γ⇓ , A⇓) , B⇓) = Typ⇓ T_2 (Γ⇓ , B⇓)
      Typ⇓ (W2 T_3) (((Γ⇓ , A⇓) , B⇓) , C⇓) = Typ⇓ T_3 ((Γ⇓ , B⇓) , C⇓)
      Typ⇓ (T ‘→’ T_1) Γ⇓ = (T⇓ : Typ⇓ T Γ⇓) → Typ⇓ T_1 (Γ⇓ , T⇓)
      Typ⇓ ‘Context’ Γ⇓ = Lifted Context
      Typ⇓ ‘Typ’ (Γ⇓ , T⇓) = Lifted (Typ (lower T⇓))
      Typ⇓ ‘Term’ (Γ⇓ , T⇓ , t⇓) = Lifted (Term (lower t⇓))
      Typ⇓ (‘Σ’ T T_1) Γ⇓ = Σ (Typ⇓ T Γ⇓) (λ T⇓ → Typ⇓ T_1 (Γ⇓ , T⇓))

      Term⇓ : ∀ {Γ : Context} {T : Typ Γ} → Term T → (Γ⇓ : Context⇓
      Term⇓ (w t) (Γ⇓ , A⇓) = Term⇓ t Γ⇓
      Term⇓ (‘λ∙’ t) Γ⇓ T⇓ = Term⇓ t (Γ⇓ , T⇓)
```

```
Term⇓ (t ‘’ₐ t₁) Γ⇓ = Term⇓ t Γ⇓ (Term⇓ t₁ Γ⇓)
Term⇓ ‘VAR₀’ (Γ⇓ , A⇓) = A⇓
Term⇓ (⌜ Γ ⌝c) Γ⇓ = lift Γ
Term⇓ (⌜ T ⌝T) Γ⇓ = lift T
Term⇓ (⌜ t ⌝t) Γ⇓ = lift t
Term⇓ ‘quote-term’ Γ⇓ (lift T⇓) = lift ⌜ T⇓ ⌝t
Term⇓ (‘quote-sigma’ {Γ₀} {Γ₁}) Γ⇓ (lift Γ , lift T) = lift (‘existT’ {Γ₁} ⌜ Γ ⌝c ⌜ T ⌝T)
Term⇓ ‘cast’ Γ⇓ T⇓ = lift (context-pick-if
  {P = Typ}
  {lower (Σ.proj₁ T⇓)}
  (W dummy)
  (lower (Σ.proj₂ T⇓)))
Term⇓ (SW t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-substTyp-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (substTyp-weakenTyp1-VAR₀ t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (weakenTyp-tProd-inv t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (weakenTyp-weakenTyp-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (substTyp1-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (weakenTyp1-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (substTyp2-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (substTyp1-substTyp-weakenTyp-inv t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (substTyp1-substTyp-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-weakenTyp-substTyp1-substTyp-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-substTyp2-substTyp1-substTyp-weakenTyp-inv t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (substTyp2-substTyp1-substTyp-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-substTyp2-substTyp1-substTyp-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (weakenTyp2-weakenTyp1 t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp1-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp1-weakenTyp-inv t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp1-weakenTyp1-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (substTyp1-weakenTyp1 t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp1-substTyp-weakenTyp1-inv t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp1-substTyp-weakenTyp1 t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-substTyp-substTyp-weakenTyp1 t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-substTyp-substTyp-weakenTyp1-inv t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (substTyp-weakenTyp1-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (substTyp3-substTyp2-substTyp1-substTyp-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-substTyp2-substTyp1-substTyp-weakenTyp1 t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (substTyp1-substTyp-tProd t) Γ⇓ T⇓ = Term⇓ t Γ⇓ T⇓
Term⇓ (substTyp2-substTyp-substTyp-weakenTyp1-weakenTyp-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (substTyp1-substTyp-weakenTyp2-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (weakenTyp-weakenTyp1-weakenTyp t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ (beta-under-subst t) Γ⇓ = Term⇓ t Γ⇓
Term⇓ ‘proj₁’’ Γ⇓ (x , p) = x
Term⇓ ‘proj₂’’ (Γ⇓ , (x , p)) = p
Term⇓ (‘existT’ x p) Γ⇓ = Term⇓ x Γ⇓ , Term⇓ p Γ⇓
Term⇓ (f ‘‘’’ x) Γ⇓ = lift (lower (Term⇓ f Γ⇓) ‘’ lower (Term⇓ x Γ⇓))
Term⇓ (f w‘‘’’ x) Γ⇓ = lift (lower (Term⇓ f Γ⇓) ‘’ lower (Term⇓ x Γ⇓))
Term⇓ (f ‘‘→’’ x) Γ⇓ = lift (lower (Term⇓ f Γ⇓) ‘→’’ lower (Term⇓ x Γ⇓))
Term⇓ (f w‘‘→’’ x) Γ⇓ = lift (lower (Term⇓ f Γ⇓) ‘→’’ lower (Term⇓ x Γ⇓))
Term⇓ (w→ x) Γ⇓ A⇓ = Term⇓ x (Σ.proj₁ Γ⇓) A⇓
Term⇓ w‘‘→’’→‘‘→’’ Γ⇓ T⇓ = T⇓
Term⇓ ‘‘→’’→w‘‘→’’ Γ⇓ T⇓ = T⇓
Term⇓ ‘tApp-nd’ Γ⇓ f⇓ x⇓ = lift (SW (lower f⇓ ‘’ₐ lower x⇓))
Term⇓ ⌜←’⌝ Γ⇓ T⇓ = T⇓
Term⇓ ⌜→’⌝ Γ⇓ T⇓ = T⇓
Term⇓ (‘‘fcomp-nd’’ {A} {B} {C}) Γ⇓ g⇓ f⇓ = lift (_‘∘’_ {ε} (lower g⇓) (lower f⇓))
Term⇓ (⌜‘’⌝ {B} {A} {b}) Γ⇓ = lift (‘λ•’ {ε} (‘VAR₀’ {ε} {_‘’_ {ε} A b}))
Term⇓ (⌜‘’⌝’ {B} {A} {b}) Γ⇓ = lift (‘λ•’ {ε} (‘VAR₀’ {ε} {_‘’_ {ε} A b}))
Term⇓ (‘cast-refl’ {T}) Γ⇓ = lift (cast-proof {T})
Term⇓ (‘cast-refl’’ {T}) Γ⇓ = lift (cast’-proof {T})

    Term⇓ (‘s→→’ {T} {B} {b} {c} {v}) Γ⇓ = lift (‘idfun’ {_‘’_ {ε}
    Term⇓ (‘s←←’ {T} {B} {b} {c} {v}) Γ⇓ = lift (‘idfun’ {_‘’_ {ε}
module well-typed-syntax-interpreter where
  open well-typed-syntax
  open well-typed-syntax-eq-dec
  max-level : Level
  max-level = well-typed-syntax-pre-interpreter.max-level

  Context⇓ : (Γ : Context) → Set (lsuc max-level)
  Context⇓ = well-typed-syntax-pre-interpreter.inner.Context⇓
    (λ ℓ P Γ’ dummy val → context-pick-if {P = P} dummy val)
    (λ ℓ P dummy val → context-pick-if-refl {P = P} {dummy})

  Typ⇓ : {Γ : Context} → Typ Γ → Context⇓ Γ → Set max-level
  Typ⇓ = well-typed-syntax-pre-interpreter.inner.Typ⇓
    (λ ℓ P Γ’ dummy val → context-pick-if {P = P} dummy val)
    (λ ℓ P dummy val → context-pick-if-refl {P = P} {dummy})

  Term⇓ : ∀ {Γ : Context} {T : Typ Γ} → Term T → (Γ⇓ : Context⇓ Γ) →
  Term⇓ = well-typed-syntax-pre-interpreter.inner.Term⇓
    (λ ℓ P Γ’ dummy val → context-pick-if {P = P} dummy val)
    (λ ℓ P dummy val → context-pick-if-refl {P = P} {dummy})
module well-typed-syntax-interpreter-full where
  open well-typed-syntax
  open well-typed-syntax-interpreter

  Contextε⇓ : Context⇓ ε
  Contextε⇓ = tt

  Typeε⇓ : Typ ε → Set max-level
  Typeε⇓ T = Typ⇓ T Contextε⇓

  Termε⇓ : {T : Typ ε} → Term T → Typeε⇓ T
  Termε⇓ t = Term⇓ t Contextε⇓

  Typeε▷⇓ : ∀ {A} → Typ (ε ▷ A) → Typeε⇓ A → Set max-level
  Typeε▷⇓ T A⇓ = Typ⇓ T (Contextε⇓ , A⇓)

  Termε▷⇓ : ∀ {A} → {T : Typ (ε ▷ A)} → Term T → (x : Typeε⇓ A) → Ty
  Termε▷⇓ t x = Term⇓ t (Contextε⇓ , x)
module löb where
  open well-typed-syntax
  open well-typed-quoted-syntax
  open well-typed-syntax-interpreter-full
  module inner (‘X’ : Typ ε) (‘f’ : Term {Γ = ε ▷ (‘□’ ‘’ ⌜ ‘X’ ⌝T)} (W ‘X’
    X : Set _
    X = Typeε⇓ ‘X’

    f’’ : (x : Typeε⇓ (‘□’ ‘’ ⌜ ‘X’ ⌝T)) → Typeε▷⇓ {‘□’ ‘’ ⌜ ‘X’ ⌝T} (W ‘X’)
    f’’ = Termε▷⇓ ‘f’

    dummy : Typ ε
    dummy = ‘Context’

    cast : (h : Σ Context Typ) → Typ (ε ▷ ‘Σ’ ‘Context’ ‘Typ’)
    cast (Γ , v) = context-pick-if {P = Typ} {Γ} (W dummy) v

    Hf : (h : Σ Context Typ) → Typ ε
```

```
Hf h = (cast h '' quote-sigma h '→'' 'X')

qh : Term {Γ = (ε ▷ 'Σ' 'Context' 'Typ')} (W ('Typ' '' 'ε'))
qh = f' w''' x
  where
    f' : Term (W ('Typ' '' ⌜ ε ▷ 'Σ' 'Context' 'Typ' ⌝c))
    f' = w→ 'cast' '''a 'VAR0'

    x : Term (W ('Term' ''1 ⌜ ε ⌝c '' ⌜ 'Σ' 'Context' 'Typ' ⌝T))
    x = (w→ 'quote-sigma' '''a 'VAR0')

h2 : Typ (ε ▷ 'Σ' 'Context' 'Typ')
h2 = (W1 '□' '' (qh w''→''' w ⌜ 'X' ⌝T))

h : Σ Context Typ
h = ((ε ▷ 'Σ' 'Context' 'Typ') , h2)

H0 : Typ ε
H0 = Hf h

H : Set
H = Term {Γ = ε} H0

'H0' : □ ('Typ' '' ⌜ ε ⌝c)
'H0' = ⌜ H0 ⌝T

'H' : Typ ε
'H' = '□' '' 'H0'

H0' : Typ ε
H0' = 'H' '→'' 'X'

H' : Set
H' = Term {Γ = ε} H0'

'H0'' : □ ('Typ' '' ⌜ ε ⌝c)
'H0'' = ⌜ H0' ⌝T

'H'' : Typ ε
'H'' = '□' '' 'H0''

toH-helper-helper : ∀ {k} → h2 ≡ k
  → □ (h2 '' quote-sigma h '→'' '□' '' ⌜ h2 '' quote-sigma h '→'' 'X' ⌝T)
  → □ (k '' quote-sigma h '→'' '□' '' ⌜ k '' quote-sigma h '→'' 'X' ⌝T)
toH-helper-helper p x = transport (λ k → □ (k '' quote-sigma h '→'' '□' '' ⌜ k '' quote-sigma h '→'' 'X' ⌝T)) p x

toH-helper : □ (cast h '' quote-sigma h '→'' 'H')
toH-helper = toH-helper-helper
  {k = context-pick-if {P = Typ} {ε ▷ 'Σ' 'Context' 'Typ'} (W dummy) h2}
  (sym (context-pick-if-refl {P = Typ} {W dummy} {h2}))
  (S00W1'→ ((''→'''→w''→''' 'o' ''fcomp-nd'' '''a ('s←←' ''o'' 'cast-refl' ''o'' ⌜→'⌝ '''a ⌜ 'λ•' 'VAR0' ⌝t)) 'o' ⌜←'⌝))

'toH' : □ ('H'' '→'' 'H')
'toH' = ⌜→'⌝ 'o' ''fcomp-nd'' '''a (⌜→'⌝ '''a ⌜ toH-helper ⌝t) 'o' ⌜←'⌝

toH : H' → H
toH h' = toH-helper 'o' h'

fromH-helper-helper : ∀ {k} → h2 ≡ k
  → □ ('□' '' ⌜ h2 '' quote-sigma h '→'' 'X' ⌝T '→'' h2 '' quote-sigma h)
  → □ ('□' '' ⌜ k '' quote-sigma h '→'' 'X' ⌝T '→'' k '' quote-sigma h)
fromH-helper-helper p x = transport (λ k → □ ('□' '' ⌜ k '' quote-sigma h '→'' 'X' ⌝T '→'' k '' quote-sigma h)) p x
```

```
fromH-helper : □ ('H' '→'' cast h '' quote-sigma h)
fromH-helper = fromH-helper-helper
  {k = context-pick-if {P = Typ} {ε ▷ 'Σ' 'Context' 'Typ'} (W dumr
  (sym (context-pick-if-refl {P = Typ} {W dummy} {h2}))
  (S00W1'← (⌜→'⌝ 'o' ''fcomp-nd'' '''a (⌜→'⌝ '''a ⌜ 'λ•' 'VAR0' ⌝t '

'fromH' : □ ('H' '→'' 'H'')
'fromH' = ⌜→'⌝ 'o' ''fcomp-nd'' '''a (⌜→'⌝ '''a ⌜ fromH-helper ⌝t) 'o'

fromH : H → H'
fromH h' = fromH-helper 'o' h'

lob : □ 'X'
lob = fromH h' '''a ⌜ h' ⌝t
  where
    f' : Term {ε ▷ '□' '' 'H0'} (W ('□' '' (⌜ '□' '' 'H0' ⌝T ''→''' ⌜ 'X
    f' = Conv0 {'H0'} {'X'} (SW1W (w∀ 'fromH' ''a 'VAR0'))

    x : Term {ε ▷ '□' '' 'H0'} (W ('□' '' ⌜ 'H' ⌝T))
    x = w→ 'quote-term' '''a 'VAR0'

    h' : H
    h' = toH ('λ•' (w→ ('λ•' 'f') '''a (w→→ 'tApp-nd' '''a f' '''a x))

lob : {'X' : Typ ε} → □ (('□' '' ⌜ 'X' ⌝T) '→'' 'X') → □ 'X'
lob {'X'} 'f' = inner.lob 'X' (un'λ•' 'f')
```

Acknowledgments

(Adam Chlipala, Matt Brown)
Acknowledgments, if needed.

References

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`.

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