

Löb's Theorem

A functional pearl of dependently typed quining

Name1
Affiliation1
Email1

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*, by Geoffrey K. Pullum
(<http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html>())

1. TODO

- cite Using Reflection to Explain and Enhance Type Theory?

2. Introduction

Löb's theorem has a variety of applications, from proving incompleteness of a logical theory as a trivial corollary, 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 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, 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 $A \rightarrow B$ are the same, if we assume A , we are also assuming $A \rightarrow B$, and

hence we have B , and since assuming A yields B , we have that $A \rightarrow B$. 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

$$\Box(\Box X \rightarrow X) \rightarrow \Box X$$

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?? 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

¹Those 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".

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.

TODO: Table of CH-Equivalence: Type \leftrightarrow set of proofs \leftrightarrow Proposition, Term/(Terminating, Well-typed, functional) Program \leftrightarrow Proof, Function Type \leftrightarrow set of functions \leftrightarrow Implication, Pairing \leftrightarrow cartesian product \leftrightarrow Conjunction, Sum Type \leftrightarrow disjoint union \leftrightarrow disjunction

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?)

3. 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 my_type: '(' + my_type + ') -> 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

```
(lambda my_type: '(' + my_type % repr(my_type) + ') -> X')
("lambda my_type: '(' + my_type %% repr(my_type) + ') -> X")
```

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 \square 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 " \rightarrow " were valid notation for function types in Python)

```
(lambda my_type:  $\square$  (my_type % repr(my_type))) -> X)
(("lambda my_type:  $\square$  (my_type %% repr(my_type)) -> X)\n (%s) % repr(my_type))
```

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 my_type: eval(my_type % repr(my_type))) -> X)
(("lambda my_type: eval(my_type %% repr(my_type)) -> X)\n (%s) % repr(my_type))
```

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 ($\square X \rightarrow X$) is a type that takes syntax trees and evaluates them; it is the type of an interpreter. (TODO: maybe move this sentence?)

4. 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 (\perp), and lambdas. (FIXME: What's the right level of simplicity?)

We begin with some standard data type declarations.

```
open import Agda.Primitive public
using (Level;  $\_ \perp \_$ ; lzero; lsuc)
```

```
infixl 1  $\_ \rightarrow \_$ 
infixr 2  $\_ \times \_$ 
infixl 1  $\_ \equiv \_$ 
```

```
record  $\top$  { $\ell$ } : Set  $\ell$  where
  constructor tt
```

```
data  $\perp$  { $\ell$ } : Set  $\ell$  where
```

```
record  $\Sigma$  { $\ell \ell'$ } (A : Set  $\ell$ ) (P : A  $\rightarrow$  Set  $\ell'$ ) : Set ( $\ell \sqcup \ell'$ ) where
  constructor  $\_ \cdot \_$ 
  field
    proj1 : A
    proj2 : P proj1
```

```
data Lifted {a b} (A : Set a) : Set (b  $\sqcup$  a) where
  lift : A  $\rightarrow$  Lifted A
```

```
lower :  $\forall$  {a b A}  $\rightarrow$  Lifted {a} {b} A  $\rightarrow$  A
lower (lift x) = x
```

```
 $\_ \times \_$  :  $\forall$  { $\ell \ell'$ } (A : Set  $\ell$ ) (B : Set  $\ell'$ )  $\rightarrow$  Set ( $\ell \sqcup \ell'$ )
A  $\times$  B =  $\Sigma$  A ( $\lambda \_ \rightarrow$  B)
```

```
data  $\equiv$  { $\ell$ } {A : Set  $\ell$ } (x : A) : A  $\rightarrow$  Set  $\ell$  where
  refl : x  $\equiv$  x
```

```
sym : {A : Set  $\ell$ }  $\rightarrow$  {x : A}  $\rightarrow$  {y : A}  $\rightarrow$  x  $\equiv$  y  $\rightarrow$  y  $\equiv$  x
sym refl = refl
```

```
trans : {A : Set}  $\rightarrow$  {x y z : A}  $\rightarrow$  x  $\equiv$  y  $\rightarrow$  y  $\equiv$  z  $\rightarrow$  x  $\equiv$  z
trans refl refl = refl
```

```
transport :  $\forall$  {A : Set} {x : A} {y : A}  $\rightarrow$  (P : A  $\rightarrow$  Set)  $\rightarrow$  x  $\equiv$  y  $\rightarrow$  P x  $\rightarrow$  P y
transport P refl v = v
```

```
mutual
  infixl 2  $\_ \triangleright \_$ 
  infixl 4  $\_ \triangleright \_$ 
```

```
data Context : Set where
   $\varepsilon$  : Context
   $\_ \triangleright \_$  : ( $\Gamma$  : Context)  $\rightarrow$  Type  $\Gamma$   $\rightarrow$  Context
```

```
data Type : Context  $\rightarrow$  Set where
  ' $\top$ ' :  $\forall$  { $\Gamma$ }  $\rightarrow$  Type  $\Gamma$ 
  (%s) :  $\forall$  { $\Gamma$ }  $\rightarrow$  Type  $\Gamma$ 
  ' $\Pi$ ' :  $\forall$  { $\Gamma$ }  $\rightarrow$  (A : Type  $\Gamma$ )  $\rightarrow$  Type ( $\Gamma \triangleright A$ )  $\rightarrow$  Type  $\Gamma$ 
```

```
data Term : { $\Gamma$  : Context}  $\rightarrow$  Type  $\Gamma$   $\rightarrow$  Set where
  'tt' :  $\forall$  { $\Gamma$ }  $\rightarrow$  Term { $\Gamma$ } ' $\top$ '
  ' $\lambda$ ' :  $\forall$  { $\Gamma$  A B}  $\rightarrow$  Term { $\Gamma \triangleright A$ } B  $\rightarrow$  Term { $\Gamma$ } (' $\Pi$ ' 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
  ⊢_⊢c : Context → Set
  ⊢_ε⊢c = ⊤
  ⊢_Γ▷ T ⊢c = Σ ⊢_Γ ⊢c ⊢ T ⊢T

  ⊢_⊢T : ∀ {Γ} → Type Γ → ⊢_Γ ⊢c → Set
  ⊢_⊢T ⊢T ⊢_Γ⊢ = ⊤
  ⊢_⊢T ⊢T ⊢_Γ⊢ = ⊥
  ⊢_⊢T A B ⊢T ⊢_Γ⊢ = (x : ⊢_A ⊢T ⊢_Γ⊢) → ⊢_B ⊢T (⊢_Γ⊢, x)

  ⊢_⊢t : ∀ {Γ} {T : Type Γ} → Term T → (⊢_Γ⊢ : ⊢_Γ ⊢c) → ⊢_T ⊢T ⊢_Γ⊢
  ⊢_⊢t ⊢T ⊢_Γ⊢ = tt
  ⊢_⊢t (λ' f) ⊢T ⊢_Γ⊢ x = ⊢_f ⊢t (⊢_Γ⊢, x)
```

5. 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 13 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`.

6. 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)

7. Trivial Encoding

```
infixr 1 _'→'_

data Type : Set where
  '→' : Type → Type → Type
  '□' : Type → Type

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

⊢_⊢ : Type → Set
⊢_A '→' B ⊢ = ⊢_A ⊢ → ⊢_B ⊢
⊢_□ T ⊢ = □ T

⊢_⊢t : ∀ {T : Type} → □ T → ⊢_T ⊢
⊢_⊢t (Löb □ 'X'→X) ⊢t = □ □ 'X'→X ⊢t (Löb □ 'X'→X)

lōb : ∀ {X} → □ ('□' X '→' X) → ⊢_X ⊢
```

`lōb f = ⊢_Lōb f ⊢t`

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

```
infixr 1 _'→'_

mutual
  data Type : Set where
    '→' : Type → Type → Type
    '□' : Type → Type
    'T' : Type
    '⊥' : Type

  data □ : Type → Set where
    Löb : ∀ {X} → □ ('□' X '→' X) → □ X
    'tt' : □ 'T'

  mutual
    ⊢_⊢ : Type → Set
    ⊢_A '→' B ⊢ = ⊢_A ⊢ → ⊢_B ⊢
    ⊢_□ T ⊢ = □ T
    ⊢_T ⊢ = ⊤
    ⊢_⊥ ⊢ = ⊥

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

  ¬_ : Set → Set
  ¬ T = T → ⊥

  '¬' : Type → Type
  '¬' T = T '→' '⊥'

  lōb : ∀ {X} → □ ('□' X '→' X) → ⊢_X ⊢
  lōb f = ⊢_Lōb f ⊢t

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

  soundness : ¬ □ '⊥'
  soundness x = ⊢_x ⊢t

  non-emptiness : □ 'T'
  non-emptiness = 'tt'
```

9. Encoding with Quines

```
module lob-by-quotes where
  infixl 2 _▷_
  infixl 3 _''_
  infixr 1 _'→'_
  infixl 3 _''_a_
  infixl 3 _w''''_a_
  infixr 2 _'o'_

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

2016/2/26


```

Term⌋ (A ww'''→''' B) Γ⌋ = lift ((lower (Term⌋ A Γ⌋)) "→" (lower (Term⌋ B Γ⌋)))
Term⌋ (A ww'''×''' B) Γ⌋ = lift ((lower (Term⌋ A Γ⌋)) "×" (lower (Term⌋ B Γ⌋)))

'other-cooperates-with' : ∀ {Γ} → Term {Γ ▷ '□' 'Bot' ▷ W ('□' 'Bot')} (W (W ('□' ('□' 'Type' Γ))))
'other-cooperates-with' {Γ} = 'eval-other' 'o' w → (w (w → (w ('λ•' 'Γ' 'VAR0' 't'))))

module inner ('X' : Type ε) ('f' : Term {ε} ('□' 'X' → 'X')) where 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 'Γ' 'fromH' 't' w'''a 'VAR0' w'''a 'Γ' 'VAR0' 't')

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

  Löb : □ 'X'
  Löb = 'fromH' ''a 'h' ''a 'Γ' 'h' 't'

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

  ⊥ _ : Type ε → Set _
  ⊥ T = Type⌋ T tt

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

  _ w'''×''' _ : ∀ {Γ X} → 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 = Term⌋ (Löb f) tt

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

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

  soundness : ¬ □ '⊥'
  soundness x = Term⌋ x tt

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

open lob

'Bot' : ∀ {Γ} → Type Γ
'Bot' {Γ} = Quine (W1 'Term' '' 'VAR0' → W1 'Term' '' 'VAR0' → W ('Type' Γ)) {- a bot takes in the source code for itself, f -}

_ cooperates-with _ : □ 'Bot' → □ 'Bot' → Type ε
b1 cooperates-with b2 = lower (Term⌋ b1 tt (lift b1) (lift b2))

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

'eval-bot' : ∀ {Γ} → Term {Γ} ('□' 'Bot' → '□' ('{- other -} Bot → Type Γ))

```

11. 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: - $X - \Box = \text{Term} \vdash H \rightarrow X$
 $\rightarrow X$ - define $y : X$ - Suppose we have a type $H \cong \text{Term} \vdash H \rightarrow X$
 \neg , and we have - $\text{toH} : \text{Term} \vdash H \rightarrow X \rightarrow H$ - $\text{fromH} : H \rightarrow \text{Term} \vdash H \rightarrow X$
 \neg - quote : $H \rightarrow \text{Term} \vdash H \rightarrow X$ - Then we can define -
 $y = (\lambda h : H. f (\text{subst} (\text{quote } h) h))$ ($\text{toH} \neg h : H. f (\text{subst} (\text{quote } h) h)$)

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

- Bibliography - Appendix - 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

2016/2/26


```
module well-typed-syntax-helpers where
```


2016/2/26

2016/2/26

```

cast'-proof {T} = cast-helper {Σ' 'Context' 'Typ'}
  {context-pick-if {P = Typ} {ε ▷ Σ' 'Context' 'Typ'} (W dummy) T}
  {T} (sym (context-pick-if-refl {P = Typ} {dummy = W dummy}))

cast-proof : ∀ {T} → Term {ε} (T " 'exist T' Γ ε ▷ Σ' 'Context' 'Typ' 'c
  '→' context-pick-if {P = Typ} (W dummy) T " 'exist T' Γ ε ▷ Σ' 'Context'
cast-proof {T} = cast-helper {Σ' 'Context' 'Typ'} {T}
  {context-pick-if {P = Typ} {ε ▷ Σ' 'Context' 'Typ'} (W dummy) T}
  (context-pick-if-refl {P = Typ} {dummy = W dummy})

'idfun' : ∀ {T} → Term {ε} (T "→" T)
'idfun' = 'λ•' 'VAR0'

mutual
Context↓ : (Γ : Context) → Set (Isuc 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↓ (T1 "x) Γ↓ = Typ↓ T1 (Γ↓, Term↓ x Γ↓)
Typ↓ (T2 "1 a) (Γ↓, A↓) = Typ↓ T2 ((Γ↓, Term↓ a Γ↓), A↓)
Typ↓ (T3 "2 a) ((Γ↓, A↓), B↓) = Typ↓ T3 ((Γ↓, Term↓ a Γ↓), A↓)
Typ↓ (T3 "3 a) (((Γ↓, A↓), B↓), C↓) = Typ↓ T3 (((Γ↓, Term↓ a Γ↓), B↓), C↓)
Typ↓ (W T1) (Γ↓, _) = Typ↓ T1 Γ↓
Typ↓ (W1 T2) ((Γ↓, A↓), B↓) = Typ↓ T2 (Γ↓, B↓)
Typ↓ (W2 T3) ((Γ↓, A↓), B↓, C↓) = Typ↓ T3 ((Γ↓, B↓), C↓)
Typ↓ (T "→" T1) Γ↓ = (T↓ : Typ↓ T Γ↓) → Typ↓ T1 (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 T1) Γ↓ = Σ (Typ↓ T Γ↓) (λ T↓ → Typ↓ T1 (Γ↓, T↓))

Term↓ : ∀ {Γ : Context} {T : Typ Γ} → Term T → (Γ↓ : Context↓ Γ)
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 Γ 'c) Γ↓ = lift Γ
Term↓ (Γ T 'T) Γ↓ = lift T
Term↓ (Γ t 't) Γ↓ = lift t
Term↓ 'quote-term' Γ↓ (lift T↓) = lift Γ T↓ t
Term↓ ('quote-sigma' {Γ0} {Γ1}) Γ↓ (lift Γ, lift T) = lift ('exist T Γ1 Γ0 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 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↓ (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↓ (weakenTyp-substTyp-substTyp-weakenTyp1 t) Γ↓ = Term
Term↓ (weakenTyp-substTyp-substTyp-weakenTyp1-inv t) Γ↓ = T
Term↓ (substTyp-weakenTyp1-weakenTyp t) Γ↓ = Term↓ t Γ↓
Term↓ (substTyp3-substTyp2-substTyp1-substTyp-weakenTyp t) Γ
Term↓ (weakenTyp-substTyp2-substTyp1-substTyp-weakenTyp t) Γ
Term↓ (substTyp1-substTyp-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (substTyp2-substTyp-substTyp-weakenTyp1-weakenTyp-w
Term↓ (substTyp1-substTyp-weakenTyp2-weakenTyp t) Γ↓ = Term
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↓ ('exist T' 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 w "→" x) Γ↓ = lift (lower (Term↓ f Γ↓) "→" lower (Term
Term↓ (f w "→" x) Γ↓ = lift (lower (Term↓ f Γ↓) "→" lower (Term
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 ← "Γ Γ↓ T↓ = T↓
Term↓ Γ → "Γ Γ↓ T↓ = T↓
Term↓ ('fcomp-nd' {A} {B} {C}) Γ↓ g↓ f↓ = lift (Γ "o" {ε} (low
Term↓ (Γ "Γ {B} {A} {b}) Γ↓ = lift ('λ•' {ε} ('VAR0' {ε} { " "
Term↓ (Γ T T {B} {A} {b}) Γ↓ = lift ('λ•' {ε} ('VAR0' {ε} { " "
Term↓ ('cast-refl' {T}) Γ↓ = lift (cast-proof {T})
Term↓ ('cast-refl' {T}) Γ↓ = lift (cast'-proof {T})
Term↓ ('s → →' {T} {B} {c} {v}) Γ↓ = lift ('idfun' { " " {ε}
Term↓ ('s ← ←' {T} {B} {b} {c} {v}) Γ↓ = lift ('idfun' { " " {ε}

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

max-level = well-typed-syntax-pre-interpretor.max-level

Context↓ : (Γ : Context) → Set (Isuc max-level)
Context↓ = well-typed-syntax-pre-interpretor.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-interpretor.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-interpretor.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})

Term↓ = well-typed-syntax-interpretor-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) → Typeε▷↓ T x
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▷('□' "Γ X' ⊢ T)}) (W X') where
    X : Set _
    X = Typeε↓ X'

    f'' : (x : Typeε↓ ('□' "Γ X' ⊢ T)) → Typeε▷↓ {'□' "Γ X' ⊢ T} (W X')
    f'' = Termε▷↓ f'

    dummy : Typ ε
    dummy = 'Context'

    cast : (Γv : 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' "Γ ε▷'Σ' 'Context' 'Typ' ⊢ c))
      f' = w→'cast' "a" 'VAR₀'

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

    h2 : Typ (ε▷'Σ' 'Context' 'Typ')
    h2 = (W1 '□' " (qh w "→" w "Γ 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' "Γ ε ⊢ c)
    'H0' = Γ H0 ⊢ T

    'H' : Typ ε
    'H' = '□' " 'H0'

    H0' : Typ ε
    H0' = 'H' '→' 'X'

    H' : Set
    H' = Term {Γ = ε} H0'

    'H0'' : □ ('Typ' "Γ ε ⊢ c)
    'H0'' = Γ H0' ⊢ T

    'H'' : Typ ε
    'H'' = '□' " 'H0''

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

    toH-helper : □ (cast h "quote-sigma h '→" 'H')
    toH-helper = toH-helper-helper

    'toH' : context-pick-if {P = Typ} {ε▷'Σ' 'Context' 'Typ'} (W dummy)
    (sym (context-pick-if-refl {P = Typ} {W dummy} {h2}))
    (S00W1→ ((Γ→'Γ' 'o' "fcomp-nd" "a" (Γ←'Γ' 'o' "cas

    xtoH : □ ('H' '→' 'H')
    'toH' = Γ→'Γ' 'o' "fcomp-nd" "a" (Γ→'Γ' "a" toH-helper ⊢ t) 'o' Γ←'Γ'

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

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

    fromH-helper : □ ('H' '→' cast h "quote-sigma h)
    fromH-helper = fromH-helper-helper

    {k = context-pick-if {P = Typ} {ε▷'Σ' 'Context' 'Typ'} (W dummy)
    (sym (context-pick-if-refl {P = Typ} {W dummy} {h2}))
    (S00W1← (Γ→'Γ' 'o' "fcomp-nd" "a" (Γ→'Γ' "a" Γλ•' 'VAR₀' ⊢ t)

    'fromH' : □ ('H' '→' 'H')
    'fromH' = Γ→'Γ' 'o' "fcomp-nd" "a" (Γ→'Γ' "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 ('□' " (Γ '□' " 'H0' ⊢ T "→" "Γ X'
      f' = Conv0 {H0'} {X'} (SW1W (w∀ 'fromH' "a" 'VAR₀'))

      x : Term {ε▷'□' " 'H0' } (W ('□' "Γ 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} → □ (('□' "Γ X' ⊢ T) '→' X') → □ X'

```

```
lob { 'X' } f = inner.lob 'X' (un'λ•' f)
```

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

P. Q. Smith, and X. Y. Jones. ...reference text...