

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* (Pullum 2000))

TODO

- cite Using Reflection to Explain and Enhance Type Theory?

1. 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_{ω}**), 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 wonderful 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 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

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".

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

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 (\perp), and `lambda`. (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
  'T' : ∀ {Γ} -> Type Γ
  '⊥' : ∀ {Γ} -> Type Γ
  'II' : ∀ {Γ} -> (A : Type Γ) -> Type (Γ ▷ A) -> Type Γ

data Term : {Γ : Context} -> Type Γ -> Set where
  'tt' : ∀ {Γ} -> Term {Γ} 'T'
  'λ' : ∀ {Γ A B} -> Term {Γ ▷ A} B -> Term ('II' 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 [Γ] = ⊥
  [ 'II' 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)
```

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 sup-

port 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 $\Box T$ to denote the type of Terms of whose syntactic type is T . We use $\ulcorner T \urcorner$ 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 ($\Box 'X' \rightarrow X$ 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:

```
lob : ∀ {X'} → □ ('□' X' '→' X') → [X']ᵀ
lob f = [Löb f]ᵀ
```

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

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

```
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 : ∀ {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 '→' '⊥'
```

```
lob : ∀ {X'} → □ ('□' X' '→' X') → [X']ᵀ
lob f = [Löb f]ᵀ
```

```
incompleteness : ¬ □ ('¬' ('□' '⊥'))
incompleteness = lob
```

```
soundness : ¬ □ '⊥'
soundness x = [x]ᵀ
```

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

8. 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
```

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 = B}))
'' : ∀ {Γ A} → Type (Γ ▷ A) → Term {Γ} A → Type Γ
'Typesε' : ∀ {Γ} → Type Γ
'□' : ∀ {Γ} → Type (Γ ▷ 'Typesε')
'→' : ∀ {Γ} → Type Γ → Type Γ → Type Γ
Quine : Type (ε ▷ 'Typesε') → Type ε
'T' : ∀ {Γ} → Type Γ
'⊥' : ∀ {Γ} → Type Γ
```

data Term : {Γ : Context} → Type Γ → Set where

```
Γ : ∀ {Γ} → Type ε → Term {Γ} 'Typesε'
Γᵀ : ∀ {Γ T} → Term {ε} T → Term {Γ} ('□' '' Γ T)
ΓᵀVAR₀ : ∀ {T} → Term {ε ▷ '□' '' Γ T} (W ('□' '' Γ '□' '' Γ T))
```

```

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} B
quine→ : ∀ {φ} → Term {ε} (Quine φ '→' φ '' Quine φ)
quine← : ∀ {φ} → Term {ε} (φ '' Quine φ '→' Quine φ)
'tt' : ∀ {Γ} → Term {Γ} 'T'
→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)
_''o_ : ∀ {Γ A B C} → Term {Γ} (B '→' C) → Term {Γ} (A '→' B) → Term {ε} C
_w''a_ : ∀ {A B T} → Term {ε ▷ T} (W ('□' '' A '→' B)) → Term {ε ▷ T} (W ('□' '' B))

□ : Type ε → Set _
□ = Term {ε}

max-level : Level
max-level = lzero

mutual
  Context↓ : (Γ : Context) → Set (lsuc max-level)
  Context↓ ε = T
  Context↓ (Γ ▷ T) = Σ (Context↓ Γ) (Type↓ {Γ} T)

Type↓ : {Γ : Context} → Type Γ → Context↓ Γ → Set max-level
Type↓ (W T) Γ↓ = Type↓ T (Σ.proj₁ Γ↓)
Type↓ (W1 T) Γ↓ = Type↓ T ((Σ.proj₁ (Σ.proj₁ Γ↓)), (Σ.proj₂ Γ↓))
Type↓ (T '' x) Γ↓ = Type↓ T (Γ↓, Term↓ x Γ↓)
Type↓ 'Typesε' Γ↓ = Lifted (Type ε)
Type↓ '□' Γ↓ = Lifted (Term {ε} (lower (Σ.proj₂ Γ↓)))
Type↓ (A '→' B) Γ↓ = Type↓ A Γ↓ → Type↓ B Γ↓
Type↓ 'T' Γ↓ = T
Type↓ '⊥' Γ↓ = ⊥
Type↓ (Quine φ) Γ↓ = Type↓ φ (Γ↓, (lift (Quine φ)))

Term↓ : ∀ {Γ : Context} {T : Type Γ} → Term T → (Γ↓ : Context↓ Γ) → Type Γ
Term↓ Γ x Γ↓ = lift x
Term↓ Γ x ⊔ Γ↓ = lift x
Term↓ Γ 'VAR₀' ⊔ Γ↓ = lift Γ (lower (Σ.proj₂ Γ↓)) ⊔
Term↓ Γ (f '' a x) Γ↓ = Term↓ f Γ↓ (Term↓ x Γ↓)
Term↓ Γ 'tt' Γ↓ = tt
Term↓ Γ (quine→ {φ}) Γ↓ x = x
Term↓ Γ (quine← {φ}) Γ↓ x = x
Term↓ Γ (λ• f) Γ↓ x = Term↓ f (Γ↓, x)
Term↓ Γ VAR₀' Γ↓ = Σ.proj₂ Γ↓
Term↓ Γ (←SW1SV→W f) Γ↓ = Term↓ f
Term↓ Γ (→SW1SV→W f) Γ↓ = Term↓ f
Term↓ Γ (w x) Γ↓ = Term↓ x (Σ.proj₁ Γ↓)
Term↓ Γ (w→ f) Γ↓ = Term↓ f (Σ.proj₁ Γ↓)
Term↓ Γ (g 'o' f) Γ↓ x = Term↓ g Γ↓ (Term↓ f Γ↓ x)
Term↓ Γ (f w''a x) Γ↓ = lift (lower (Term↓ f Γ↓) '' a lower (Term↓ x Γ↓))

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

  'toH' : □ (('□' '' 'H' '→' 'X') '→' 'H')

  'toH' = ←SW1SV→W quine←
  'fromH' = →SW1SV→W quine→
  'h' : Term 'H'
  'h' = 'toH' ''a (f 'o' '□' 'H'→'X')

  Löb : □ 'X'
  Löb = 'fromH' ''a 'h' ''a 'h' ⊔

  Lb : ∀ {X} → Type {ε} C
  Lb = Term {ε} C
  Löb f = Term {ε} C

  _ : Type ε → Set _
  [T] = Type↓ T tt

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

  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'

```

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

```

module prisoners-dilemma where

module lob where
  infixl 2 _▷_
  infixl 3 _''_
  infixr 1 _'→'_
  infixr 1 _''→''_
  infixr 1 _ww''→''_
  infixl 3 _''a_
  infixl 3 _w''a_
  infixr 2 _''o'_
  infixr 2 _''x'_
  infixr 2 _''x''_
  infixr 2 _w''x''_

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

```

2016/2/27

$$\begin{aligned} \text{transport} &: \forall \{A : \text{Set}\} \{x : A\} \{y : A\} \rightarrow (P : A \rightarrow \text{Set}) \\ &\rightarrow x \equiv y \rightarrow P x \rightarrow P y \\ \text{transport } P \text{ refl } v &= v \end{aligned}$$
$$\begin{array}{l} \text{---}'' : \forall \{ \Gamma A \} \rightarrow \text{Typ} (\Gamma \triangleright A) \rightarrow \text{Term} \{ \Gamma \} A \rightarrow \text{Typ} \Gamma \\ \text{---}''_1 : \forall \{ \Gamma A B \} \rightarrow (C : \text{Typ} (\Gamma \triangleright A \triangleright B)) \rightarrow (a : \text{Term} \\ \text{---}''_2 : \forall \{ \Gamma A B C \} \rightarrow (D : \text{Typ} (\Gamma \triangleright A \triangleright B \triangleright C)) \rightarrow (a \end{array}$$
$$\begin{aligned}
& \vdash : \forall \{ \Gamma A B \} \rightarrow \text{Term } \{ \Gamma \} B \rightarrow \text{Term } \{ \Gamma = \Gamma \triangleright A \} (W \{ \Gamma = \Gamma \} B) \\
& \text{'}\lambda\bullet\text{'}: \forall \{ \Gamma A B \} \rightarrow \text{Term } \{ \Gamma = (\Gamma \triangleright A) \} B \rightarrow \text{Term } \{ \Gamma \} (A \text{'}\rightarrow\text{' } B) \\
& \text{'}\text{'}_a\text{'}\text{'}: \forall \{ \Gamma A B \} \rightarrow (f : \text{Term } \{ \Gamma \} (A \text{'}\rightarrow\text{' } B)) \rightarrow (x : \text{Term } \{ \Gamma \} A) \\
& \text{'}\text{VAR}_0\text{'}: \forall \{ \Gamma T \} \rightarrow \text{Term } \{ \Gamma = \Gamma \triangleright T \} (W T) \\
& \text{'}_c\text{'}: \forall \{ \Gamma \} \rightarrow \text{Context} \rightarrow \text{Term } \{ \Gamma \} \text{'Context'} \\
& \text{'}_T\text{'}: \forall \{ \Gamma \Gamma' \} \rightarrow \text{Typ } \Gamma' \rightarrow \text{Term } \{ \Gamma \} (\text{'Typ' } \text{'}_c\text{' } \Gamma' \text{'}_c\text{'}) \\
& \text{'}_T\text{'}: \forall \{ \Gamma \Gamma' \} \{ T : \text{Typ } \Gamma' \} \rightarrow \text{Term } T \rightarrow \text{Term } \{ \Gamma \} (\text{'Term' } \text{'}_1\text{' } \Gamma' \\
& \text{'quote-term' } : \forall \{ \Gamma \Gamma' \} \{ A : \text{Typ } \Gamma' \} \rightarrow \text{Term } \{ \Gamma \} (\text{'Term' } \text{'}_1\text{' } \Gamma' \\
& \text{'quote-sigma' } : \forall \{ \Gamma \Gamma' \} \rightarrow \text{Term } \{ \Gamma \} (\text{'}\Sigma\text{' } \text{'Context' } \text{'Typ' } \text{'}\rightarrow\text{' } W \\
& \text{'cast' } : \text{Term } \{ \varepsilon \} (\text{'}\Sigma\text{' } \text{'Context' } \text{'Typ' } \text{'}\rightarrow\text{' } W (\text{'Typ' } \text{'}_c\text{' } \varepsilon \triangleright \text{'}\Sigma\text{' } \text{'Co} \\
& \text{SW} : \forall \{ \Gamma A B \} \{ a : \text{Term } \{ \Gamma \} A \} \rightarrow \text{Term } \{ \Gamma \} (W B \text{'}_a\text{'}) \rightarrow \text{Term} \\
& \text{weakenTyp-substTyp-tProd} : \forall \{ \Gamma T T' A B \} \{ a : \text{Term } \{ \Gamma \} T \} \rightarrow \text{Term} \\
& \text{substTyp-weakenTyp1-VAR}_0 : \forall \{ \Gamma A T \} \rightarrow \text{Term } \{ \Gamma \triangleright A \} (W1 T \text{'}_a\text{'}) \\
& \text{weakenTyp-tProd} : \forall \{ \Gamma A B C \} \rightarrow \text{Term } \{ \Gamma = \Gamma \triangleright C \} (W (A \text{'}\rightarrow\text{' } B \\
& \text{weakenTyp-tProd-inv} : \forall \{ \Gamma A B C \} \rightarrow \text{Term } \{ \Gamma = \Gamma \triangleright C \} (W A \text{'}\rightarrow\text{' } B \\
& \text{weakenTyp-weakenTyp-tProd} : \forall \{ \Gamma A B C D \} \rightarrow \text{Term } \{ \Gamma \triangleright C \triangleright D \\
& \text{substTyp1-tProd} : \forall \{ \Gamma T T' A B \} \{ a : \text{Term } \{ \Gamma \} T \} \rightarrow \text{Term } \{ \Gamma \triangleright T \\
& \text{weakenTyp1-tProd} : \forall \{ \Gamma C D A B \} \rightarrow \text{Term } \{ \Gamma \triangleright C \triangleright W D \} (W1 \\
& \text{substTyp2-tProd} : \forall \{ \Gamma T T' T'' A B \} \{ a : \text{Term } \{ \Gamma \} T \} \rightarrow \text{Term } \{ \Gamma \\
& \text{substTyp1-substTyp-weakenTyp-inv} : \forall \{ \Gamma C T A \} \{ a : \text{Term } \{ \Gamma \} C \\
& \text{substTyp1-substTyp-weakenTyp} : \forall \{ \Gamma C T A \} \{ a : \text{Term } \{ \Gamma \} C \} \{ C \\
& \text{weakenTyp-weakenTyp-substTyp1-substTyp-weakenTyp} : \forall \{ \Gamma C T \\
& \text{weakenTyp-substTyp2-substTyp1-substTyp-weakenTyp-inv} : \forall \{ \Gamma \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T' \} (W (T \text{'}_1\text{' } a \text{'}_b\text{'})) \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T' \} (W (W T \text{'}_2\text{' } a \text{'}_1\text{' } b \text{'}_c\text{'})) \\
& \text{substTyp2-substTyp1-substTyp-weakenTyp} : \forall \{ \Gamma A B C T \} \{ a : \text{Term} \\
& \rightarrow \text{Term } \{ \Gamma \} (W T \text{'}_2\text{' } a \text{'}_1\text{' } b \text{'}_c\text{'}) \\
& \rightarrow \text{Term } \{ \Gamma \} (T \text{'}_1\text{' } a \text{'}_b\text{'}) \\
& \text{weakenTyp-substTyp2-substTyp1-substTyp-tProd} : \forall \{ \Gamma T T' T'' T''' \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T''' \} (W ((A \text{'}\rightarrow\text{' } B) \text{'}_2\text{' } a \text{'}_1\text{' } b \text{'}_c\text{'})) \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T''' \} ((W (A \text{'}_2\text{' } a \text{'}_1\text{' } b \text{'}_c\text{'})) \text{'}\rightarrow\text{' } (W1 (B \text{'}_3\text{' } a \text{'}_2\text{' } b \\
& \text{weakenTyp2-weakenTyp1} : \forall \{ \Gamma A B C D \} \rightarrow \text{Term } \{ \Gamma \triangleright A \triangleright W B \triangleright \\
& \text{weakenTyp1-weakenTyp} : \forall \{ \Gamma A B C \} \rightarrow \text{Term } \{ \Gamma \triangleright A \triangleright W B \} (W \\
& \text{weakenTyp1-weakenTyp-inv} : \forall \{ \Gamma A B C \} \rightarrow \text{Term } \{ \Gamma \triangleright A \triangleright W B \} \\
& \text{weakenTyp1-weakenTyp1-weakenTyp} : \forall \{ \Gamma A B C T \} \rightarrow \text{Term } \{ \Gamma \\
& \text{substTyp1-weakenTyp1} : \forall \{ \Gamma A B C \} \{ a : \text{Term } \{ \Gamma \} A \} \rightarrow \text{Term} \\
& \text{weakenTyp1-substTyp-weakenTyp1-inv} : \forall \{ \Gamma A T' T' T'' \} \{ a : \text{Term} \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T'' \triangleright W (T' \text{'}_a\text{'}) \} (W1 (W (T'' \text{'}_a\text{'}))) \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T'' \triangleright W (T' \text{'}_a\text{'}) \} (W1 (W T \text{'}_1\text{' } a)) \\
& \text{weakenTyp1-substTyp-weakenTyp1} : \forall \{ \Gamma A T'' T' T'' \} \{ a : \text{Term } \{ \Gamma \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T'' \triangleright W (T' \text{'}_a\text{'}) \} (W1 (W T \text{'}_1\text{' } a)) \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T'' \triangleright W (T' \text{'}_a\text{'}) \} (W1 (W (T'' \text{'}_a\text{'}))) \\
& \text{weakenTyp-substTyp-substTyp-weakenTyp1} : \forall \{ \Gamma T' B A \} \{ b : \text{Term} \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T' \} (W (W1 T'' \text{'}_a\text{' } b)) \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T' \} (W (T'' (SW ((\lambda\bullet\text{' } a) \text{'}_a\text{' } b)))) \\
& \text{weakenTyp-substTyp-substTyp-weakenTyp1-inv} : \forall \{ \Gamma T' B A \} \{ b : \text{Term} \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T' \} (W (T'' (SW ((\lambda\bullet\text{' } a) \text{'}_a\text{' } b)))) \\
& \rightarrow \text{Term } \{ \Gamma \triangleright T' \} (W (W1 T'' \text{'}_a\text{' } b)) \\
& \text{substTyp-weakenTyp1-weakenTyp} : \forall \{ \Gamma T \} \{ A : \text{Typ } \Gamma \} \{ B : \text{Typ } \Gamma \\
& \rightarrow \text{Term } \{ \Gamma \triangleright B \} (\text{'}_a\text{'} : \text{Term } \{ \Gamma = \Gamma \triangleright T \} (W \{ \Gamma = \Gamma \} \{ A = T \} B)) \\
& \rightarrow \text{Typ } \{ \Gamma \triangleright B \} (\text{'}_a\text{'} : \text{Term } \{ \Gamma = \Gamma \triangleright T \} (W \{ \Gamma = \Gamma \} \{ A = T \} B))
\end{aligned}$$


```

→ Term {Γ = Γ ▷ T} (W1 (W A) " a)
→ Term {Γ = Γ ▷ T} (W A)
substTyp3-substTyp2-substTyp1-substTyp-weakenTyp : ∀ {Γ A B C D T T'} {a : Term {Γ = Γ ▷ T}} {b : Term {Γ = Γ ▷ T'}} {c : Term {Γ = Γ ▷ T}} {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 C T T'} {a : Term {Γ = Γ ▷ T}} {b : Term {Γ = Γ ▷ T'}} {c : Term {Γ = (Γ ▷ T')}} (W (W1 T "₂ a "₁ b " substTyp1-substTyp-weakenTyp1 (W (W T "₃ a "₂ b "₁ c) " c))
→ Term {Γ = (Γ ▷ T')} (W (W1 T "₂ a "₁ b " substTyp1-substTyp-weakenTyp1 (W (W T "₃ a "₂ b "₁ c) " c))
→ Term {Γ = (Γ ▷ T')} (W (T "₁ a " c))
substTyp1-substTyp-tProd : ∀ {Γ T T' A B a b} → Term ((_ '→'_ {Γ = Γ ▷ T} (W (W T "₃ a "₂ b "₁ c) " c))
substTyp2-substTyp-substTyp-weakenTyp1-weakenTyp-weakenTyp : ∀ {Γ A B C T T'} {a : Term {Γ = Γ ▷ T}} {b : Term {Γ = Γ ▷ T'}} {c : Term {Γ = (Γ ▷ T')}} (W (C " a))
→ Term {Γ = (Γ ▷ T')} (W1 (W (W T "₃ a "₂ b "₁ c) " c))
→ Term {Γ = (Γ ▷ T')} (W (T " a))
substTyp1-substTyp-weakenTyp2-weakenTyp : ∀ {Γ T' A B T} {a : Term {Γ = Γ ▷ T}} {b : Term {Γ = Γ ▷ T'}} (W1 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 ▷ A ▷ W B ▷ (SW C) (W (W T "₃ a "₂ b "₁ c) " c))
beta-under-subst : ∀ {Γ A B B'} {g : Term {Γ}} {A '→' W B} {x : Term {Γ} A} " SW ('quote-sigma' " a 'existT' Γ ε ▷ 'Σ' 'Context' 'Typ' Γ T T')
→ Term (B' " SW (g " a x))
'proj₁' : ∀ {Γ} {T : Typ Γ} {P : Typ (Γ ▷ T)} → Term ('Σ' T P '→' W T) ((SW ('cast' " a 'existT' Γ ε ▷ 'Σ' 'Context' 'Typ' Γ T T')
'proj₂' : ∀ {Γ} {T : Typ Γ} {P : Typ (Γ ▷ T)} → Term {Γ ▷ 'Σ' T P} (W1 P " SW ('SW (weakenTyp1-weakenTyp (substTyp-ContextTyp)
'existT' : ∀ {Γ T P} (x : Term {Γ} T) (p : Term (P " x)) → Term ('Σ' T P) "→"
{- these are redundant, but useful for not having to normalize the subsequent ones -}
" '→' : ∀ {Γ} {A : Typ Γ}
→ Term {ε} ('Typ' " Γ ▷ A Γ c)
→ Term {ε} ('Term' "₁ Γ Γ c " Γ A T)
→ Term {ε} ('Typ' " Γ Γ c)
_w " : ∀ {X Γ} {A : Typ Γ}
→ Term {ε ▷ X} (W ('Typ' " Γ ▷ A Γ c))
→ Term {ε ▷ X} (W ('Term' "₁ Γ Γ c " Γ A T))
→ Term {ε ▷ X} (W ('Typ' " Γ Γ c))
"→" : ∀ {Γ}
→ Term {ε} ('Typ' " Γ)
→ Term {ε} ('Typ' " Γ)
→ Term {ε} ('Typ' " Γ)
_w "→" : ∀ {X Γ}
→ Term {ε ▷ X} (W ('Typ' " Γ))
→ Term {ε ▷ X} (W ('Typ' " Γ))
→ Term {ε ▷ X} (W ('Typ' " Γ))
module well-typed-syntax-helpers where
w→ : ∀ {Γ A B C} → Term (A '→' W B) → Term {Γ = Γ ▷ C} (W (w→ W B))
{- things that were postulates, but are no longer -}
"→" : ∀ {T}
{b : Term {ε} ('Typ' " Γ ε Γ c)}
{c : Term {ε ▷ T} (W ('Typ' " Γ ε Γ c))}
{e : Term {ε} T}
→ Term {ε} ('Term' "₁ Γ ε Γ c " (SW ('λ•' (c w→ " w b) " a e)
"→" W ('Term' "₁ Γ ε Γ c " (SW ('λ•' c " a e) "→" b)))
w "→" : ∀ {T}
{b : Term {ε} ('Typ' " Γ ε Γ c)}
{c : Term {ε ▷ T} (W ('Typ' " Γ ε Γ c))}
{e : Term {ε} T}
→ Term {ε} ('Term' "₁ Γ ε Γ c " (SW ('λ•' c " a e) "→" b)
"→" W ('Term' "₁ Γ ε Γ c " (SW ('λ•' (c w→ " w b) " a e) "→" b)))
'tApp-nd' : ∀ {Γ} {A : Term {ε} ('Typ' " Γ)} {B : Term {ε} ('Typ' " Γ)} →
Term {ε} ('Term' "₁ Γ " (A "→" B)
"→" W ('Term' "₁ Γ " A
"→" W ('Term' "₁ Γ " B)))
← : ∀ {H X} →
Term {ε} ('Term' "₁ Γ ε Γ c " (Γ H T "→" Γ X T)
"→" W ('Term' "₁ Γ ε Γ c " Γ H '→' W X T))

```



```

substTyp-tProd : ∀ {Γ T A B} {a : Term {Γ} T} →
  Term {Γ} ((A '→' B) " a)
  → Term {Γ} ( _ '→' _ {Γ = Γ} (A " a) (B " a))
substTyp-tProd {Γ} {T} {A} {B} {a} x = SW ((WSV (w x)) 't' a)

SV = 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 → Term {Γ = Γ▷C} (W A '→' W B)
SW1V = substTyp-weakenTyp1-VAR0

S1V : ∀ {Γ T T' A B} {a : Term {Γ} T} → Term {Γ▷T' " a} ((A '→' B) " a) → Term {Γ▷T' " a} ((A " a) '→' (B " a))
S1V = 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)
wV = weakenProd

w1 : ∀ {Γ A B C} → Term {Γ = Γ▷B} C → Term {Γ = Γ▷A▷B} {Γ = Γ} {A = A} B (W1 {Γ = Γ} {A = A} {B = B} C)
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} A) → Term {Γ▷A▷B} (C " a)
f 't'1 x = un'λ•' (SV ('λ•' ('λ•' f) " a x))
_ 't'2 _ : ∀ {Γ A B C D} → (c : Term {Γ = Γ▷A▷B▷C} D) → (a : Term {Γ} A) → Term {Γ▷A▷B▷C} (D " a)
f 't'2 x = un'λ•' (S1V (un'λ•' (SV ('λ•' ('λ•' ('λ•' f) " a x))))

S10W' : ∀ {Γ C T A} {a : Term {Γ} C} {b : Term {Γ} (T " a)} → Term {Γ▷W B " a} (W1 C " a)
S10W' = substTyp1-substTyp-weakenTyp-inv

S10W : ∀ {Γ C T A} {a : Term {Γ} C} {b : Term {Γ} (T " a)} → Term {Γ▷W B " a} (W1 C " a)
S10W = substTyp1-substTyp-weakenTyp

substTyp1-substTyp-weakenTyp-weakenTyp : ∀ {Γ T A} {B : Typ (Γ▷A▷B)} {a : Term {Γ} A}
  → {b : Term {Γ} (B " a)}
  → Term {Γ} (W (W T) " 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 : Term {Γ} (C " a " b)}
  → Term {Γ} (W T " a " b)
  → Term {Γ} (T " a " b)
S210W = substTyp2-substTyp1-substTyp-weakenTyp

substTyp2-substTyp1-substTyp-weakenTyp-weakenTyp : ∀ {Γ A B C T} {a : Term {Γ} A} {b : Term {Γ} (B " a)} {c : Term {Γ} (C " a " b)}
  → Term {Γ} (W (W T) " a " b " c)
  → Term {Γ} (T " a)
substTyp2-substTyp1-substTyp-weakenTyp-weakenTyp x = S10W (S210W x)

S210WW = substTyp2-substTyp1-substTyp-weakenTyp-weakenTyp

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

W1W1W : ∀ {Γ A B C T} → Term {Γ▷A▷B▷W} (W C) (W1 (W1 (W C)))
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 x) (A '→' B))))

weakenProd-nd : ∀ {Γ A B C D} →
  Term {Γ = Γ▷D} (W (A '→' B) " C)
  → Term {Γ = Γ▷D} (W A " W B " W C)
weakenProd-nd x = 'λ•' (weakenTyp-tProd-inv ('λ•' (W1W (SW1V (un'λ•' (weakenTyp-tProd (w x) (A '→' B) " C))))))

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•' (W1W (SW1V (un'λ•' (weakenTyp-tProd (w x) (A '→' B) " C))))))

weakenProd-nd-Prod-nd : ∀ {Γ A B C D} →
  Term (A '→' B) " C
  → Term {Γ▷A▷B▷C} (D " a)
weakenProd-nd-Prod-nd x = weakenTyp-tProd-nd (w x)

S1W1 : ∀ {Γ A B C} {a : Term {Γ} A} → Term {Γ▷W B " a} (W1 C " a)
S1W1 = substTyp1-substTyp-weakenTyp-inv

S1W : ∀ {Γ A B C} {a : Term {Γ} A} → Term {Γ▷W B " a} (W1 C " a)
S1W = S1W1

S1W' : ∀ {Γ A B C} {a : Term {Γ} A} → Term {Γ▷W B " a} (W1 C " a)
S1W' = S1W1

S1W1W : ∀ {Γ T' B A} {b : Term {Γ} B} {a : Term {Γ▷B} (W A)} → Term {Γ▷T'} (W (W1 T' " a " b))
S1W1W = weakenTyp-substTyp-substTyp-weakenTyp1

WS00W1 : ∀ {Γ T' B A} {b : Term {Γ} B} {a : Term {Γ▷B} (W A)} → Term {Γ▷T'} (W (W1 T' " a " b))
WS00W1 = S1W1W

WS00W : ∀ {Γ T' B A} {b : Term {Γ} B} {a : Term {Γ▷B} (W A)} → Term {Γ▷T'} (W (W1 T' " a " b))
WS00W = WS00W1

WS00W1 : ∀ {Γ T' B A} {b : Term {Γ} B} {a : Term {Γ▷B} (W A)} → Term {Γ▷T'} (W (W1 T' " a " b))
WS00W1 = WS00W1

```

```

WS00W1' : ∀ {Γ T' B A} {b : Term {Γ} B} {a : Term {Γ ▷ B} (W A)} {T : Typ (Γ ▷ A)} {X} →
  → 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' (un'λ•' x))

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} {c : Term {Γ = (Γ ▷ T')} (W (C " a))}
  → Term {Γ = (Γ ▷ T')} (W1 (W (W T) " a " b) " c)
  → Term {Γ = (Γ ▷ T')} (W (T' a))
S200W1WW = substTyp2-substTyp-substTyp-weakenTyp1-weakenTyp-weakenTyp

S10W2W : ∀ {Γ T' A B T} {a : Term {Γ ▷ T'} (W A)} {b : Term {Γ ▷ T'} (W1 B " a)}
  → Term {Γ ▷ T'} (W2 (W T) " 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-syntax-eq-dec where
  open well-typed-syntax
  open well-typed-syntax-helpers
  open well-typed-syntax-context-helpers

'ε' : Term {Γ = ε} 'Context'
'ε' = 'Γ ε' ⊢ c

'□' : Typ (ε ▷ 'Typ' "' ε')
'□' = 'Term' "'_1 ε'

module well-typed-syntax 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 _"o"_

quote-sigma : (Γ v : Σ Context Typ) → Term {ε} ('Σ' 'Context' 'Typ')
quote-sigma (Γ , v) = 'exist T' 'Γ Γ' ⊢ c Γ v ⊢ T

_ "o" _ : ∀ {A B C}
  → □ ('□' "' (C "→"' B))
  → □ ('□' "' (A "→"' C))
  → □ ('□' "' (A "→"' B))
g "'o" f = ('fcomp-nd' "'a f"'a g)

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

S10W2W : ∀ {Γ T' A B T} {a : Term {Γ ▷ T'} (W A)} {b : Term {Γ ▷ T'} (W1 B " a)}
  → Term {Γ ▷ T'} (W2 (W T) " a " b)
  → Term {Γ ▷ T'} (W1 T' a)
S10W2W = substTyp1-substTyp-weakenTyp2-weakenTyp

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 val
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 '→'' A)
cast-helper refl = 'λ•' 'VAR0'

cast'-proof : ∀ {T} → Term {ε} (context-pick-if {P = Typ} (W dummy)
'→'' T'' 'exist T' Γ ε ▷ 'Σ' 'Context' 'Typ' 'c' Γ T 'T)
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' 'Typ' 'c'
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↓), B↓)
Typ↓ (T3 ''3 a) (((Γ↓, A↓), B↓), C↓) = Typ↓ T3 (((Γ↓, Term↓ a Γ↓), A↓), (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' Γ0 Γ1)
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↓ (weaken Typ-subst Typ-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (subst Typ-weaken Typ1-VAR0 t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (weaken Typ-tProd-inv t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (weaken Typ-weaken Typ-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (subst Typ1-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (weaken Typ1-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (subst Typ2-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (subst Typ1-subst Typ-weaken Typ-inv t) Γ↓ = Term↓ t Γ↓
Term↓ (subst Typ1-subst Typ-weaken Typ t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ-weaken Typ-subst Typ1-subst Typ-weaken Typ t) Γ↓ = Term↓ t Γ↓
Term↓ (subst Typ2-subst Typ1-subst Typ-weaken Typ t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ-subst Typ2-subst Typ1-subst Typ-tProd t) Γ↓ T↓ = Term↓ t Γ↓
Term↓ (weaken Typ2-weaken Typ1 t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ1-weaken Typ t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ1-weaken Typ-inv t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ1-weaken Typ1-weaken Typ t) Γ↓ = Term↓ t Γ↓
Term↓ (subst Typ1-weaken Typ1 t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ1-subst Typ-weaken Typ1-inv t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ1-subst Typ-weaken Typ1 t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ-subst Typ-subst Typ-weaken Typ1 t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ-subst Typ-subst Typ-weaken Typ1-inv t) Γ↓ = Term↓ t Γ↓
Term↓ (subst Typ3-subst Typ2-subst Typ1-subst Typ-weaken Typ t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ-subst Typ2-subst Typ1-subst Typ-weaken Typ1 t) Γ↓ = Term↓ t Γ↓
Term↓ (subst Typ1-subst Typ-tProd t) Γ↓ T↓ = Term↓ t Γ↓ T↓
Term↓ (subst Typ2-subst Typ-subst Typ-weaken Typ1-weaken Typ-weaken Typ t) Γ↓ = Term↓ t Γ↓
Term↓ (weaken Typ-weaken Typ1-weaken Typ 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↓ 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 (_ 'o' _ {ε}) (lower g↓ ''a lower f↓)
Term↓ (Γ'' {B} {A} {b}) Γ↓ = lift ('λ•' {ε}) ('VAR0' {ε} {'' _ '' _})
Term↓ (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} {b} {c} {v}) Γ↓ = lift ('idfun' { _ " _ {ε} (lower (Term↓ b tt (Term↓ v Γ↓))) (lower (Term↓ c tt (Term↓ v Γ↓)))
Term↓ ('s←←' {T} {B} {b} {c} {v}) Γ↓ = lift ('idfun' { _ " _ {ε} (lower (Term↓ b tt (Term↓ v Γ↓))) (lower (Term↓ c tt (Term↓ v Γ↓)))
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) → Typ↓ T 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↓ : Contextε ↓
  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')) (W dummy)
    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 : Σ Context Typ) → Typ ε

    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

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

    toH' : □ ('H' '→' 'H')
    'toH' = Γ→'→' 'o' "fcomp-nd" " "a (Γ→'→' " "a toH-helper ⊔t) 'o' 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 :  $\square$  ('H'  $\rightarrow$ '' cast h '' quote-sigma h)
fromH-helper = fromH-helper-helper
  {k = context-pick-if {P = Typ} { $\varepsilon \triangleright$  'Σ' 'Context' 'Typ'} (W dummy) h2}
  (sym (context-pick-if-refl {P = Typ} {W dummy} {h2}))
  (S00W1'  $\leftarrow$  ( $\Gamma \rightarrow$ '' 'o' ''fcomp-nd'' ''a ( $\Gamma \rightarrow$ '' ''a  $\Gamma$  'λ•' 'VAR0'  $\neg$ t ''o'' 'cast-refl'' ''o'' 's $\rightarrow$  $\rightarrow$ '') 'o' w''  $\rightarrow$ ''  $\rightarrow$ ''  $\rightarrow$ '')))

'fromH' :  $\square$  ('H'  $\rightarrow$ '' 'H'')
'fromH' =  $\Gamma \rightarrow$ '' 'o' ''fcomp-nd'' ''a ( $\Gamma \rightarrow$ '' ''a  $\Gamma$  fromH-helper  $\neg$ t) 'o'  $\Gamma \leftarrow$ ''

fromH : H  $\rightarrow$  H'
fromH h' = fromH-helper 'o' h'

lob :  $\square$  'X'
lob = fromH h' ''a  $\Gamma$  h'  $\neg$ t
  where
    f' : Term { $\varepsilon \triangleright$  '□' '' 'H0'} (W ('□' '' ( $\Gamma$  '□' '' 'H0'  $\neg$ T '' $\rightarrow$ ''  $\Gamma$  'X'  $\neg$ T)))
    f' = Conv0 {'H0'} {'X'} (SW1W (w $\forall$  'fromH' ''a 'VAR0'))

    x : Term { $\varepsilon \triangleright$  '□' '' 'H0'} (W ('□' ''  $\Gamma$  'H'  $\neg$ T))
    x = w $\rightarrow$  'quote-term' ''a 'VAR0'

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

lob : {'X' : Typ  $\varepsilon$ }  $\rightarrow$   $\square$  (('□' ''  $\Gamma$  'X'  $\neg$ T) ' $\rightarrow$ '' 'X')  $\rightarrow$   $\square$  '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

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