Explicit weakening

Citation for published version:
Wadler, P 2024, Explicit weakening. in A Bieniusa, M Degen & S Wehr (eds), *A Second Soul: Celebrating the Many Languages of Programming - Festschrift in Honor of Peter Thiemann's Sixtieth Birthday.* Electronic Proceedings in Theoretical Computer Science, vol. 413, Open Publishing Association, pp. 15-26, A Second Soul, Freiburg, Baden-Württemberg, Germany, 30/08/24. https://doi.org/10.4204/EPTCS.413.2

Digital Object Identifier (DOI):
10.4204/EPTCS.413.2

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
A Second Soul

Explicit Weakening

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

I present a novel formulation of substitution, where facts about substitution that previously required tens or hundreds of lines to justify in a proof assistant now follow immediately—they can be justified by writing the four letters "refl". The paper is an executable literate Agda script, and source of the paper is available as an artifact in the file

Weaken.lagda.md

Not all consequences of the pandemic have been awful. For the last three years, I've had the great pleasure of meeting with Peter Thiemann and Jeremy Siek for a couple of hours every week, via Zoom, exploring topics including core calculi, gradual typing, and formalisation in Agda. The work reported here arose from those discussions, and is dedicated to Peter on the occasion of his 60th birthday.

1 Introduction

Every user of a proof assistant has suffered the plague of lemmas: sometimes a fact that requires zero lines of justification for a pen-and-paper proof may require tens or hundreds of lines of justification when using a proof assistant.

Properties of substitution often give rise to such an inundation of lemmas. To give one example—the one which motivated this work—I recall a proof formalised in Agda of the gradual guarantee for a simply-typed gradual cast calculus. The calculus uses lambda terms with de Bruijn indices, and the proof depends crucially on the following equivalence.

```
(N ↑) [ M ]₀  ≡  N                                                    (*)
```

Here `N [ M ]₀` substitutes term `M` for de Bruijn index zero in term `N`, while `N ↑` increments by one every free de Bruijn index in term `N`. Hence `N ↑` will not contain de Bruijn index zero, and the equation appears obvious. However, my formal proof required eleven lemmas and 96 non-comment lines of code to establish the above fact.

Here I present a novel formulation of substitution where equation (*) is obvious to the proof assistant as well as to the person conducting the proof: it holds by definitional equality. Further, many other equations that one needs also hold by definition and much of the remaining work can be handled by automatically applied rewrites. As a result, the facts about substitution that we need can be proved trivially. In my proof assistant of choice, Agda, one needs to write just four letters, `refl`, denoting proof by reflexivity.

Since a single example may fail to convince, I give two more.

First, in the proof of the gradual guarantee mentioned earlier, it is not just the above equation but *every* property of substitution in the proof that is rendered trivial by the new formulation given here.

Second, consider an example from the textbook *Programming Language Foundations* in Agda, by Kokke, Siek, and Wadler [11, 7], henceforth PLFA. Chapter Substitution is devoted to proving the following equation, for terms using de Bruijn indexes.

```
N [ M ]₀ [ L ]₀ ≡ N [ L ]₁ [ M [ L ]₀ ]₀
```

Here `N [ M ]₀`, as before, substitutes term `M` for de Bruijn index zero in term `N`, while `N [ L ]₁` substitutes term `L` for de Bruijn index one in term `N`. The equation states that we can substitute in either order: we can first substitute `M` into `N` and then substitute `L` into the result, or we can first substitute `L` into `N` and then substitute `M` (adjusted by substituting `L` into `M`) into the result. The chapter takes several hundred lines to achieve the above result, and was considered so long and tedious that it was relegated to an appendix. With the new formulation, the result becomes immediate.

My formulation is inspired by the explicit substitutions of Abadi, Cardelli, Curien, and Levy [1, 2], henceforth ACCL, one of the seminal works in the area. ACCL wished to devise a calculus that could aid in the design of implementations, while we focus on support for automated reasoning; and ACCL considered applications to untyped, simply-typed, and polymorphic lambda calculus (System F), while we focus on simply-typed lambda calculus. A vast body of literature is devoted to explicit substitution. Helpful surveys include Kesner [6] and Rose, Bloo, and Lang [9]. Another inspiration for this work is the Autosubst system of Schafer, Tebbi, and Smolka [10].

Despite the large number of variations that have been considered, I have not found a formulation that matches the one given here. The closest are David and Guillaume [4] and Hendricks and van Oostrom [5] both of whom use an explicit weakening, but with quite different goals. In particular, both used named variables and single substitution, whereas de Bruijn variables and simultaneous substitution are crucial to both ACCL and the approach taken here.

The formulation given here is couched in terms of de Bruijn indices and intrinsic types, as first proposed by Altenkirch and Reus [3]. Lambda calculus is typically formulated using named variables and extrinsic typing rules, but when using a proof assistant it is often more convenient to use de Bruijn indices and intrinsic typing rules. With named variables the Church numeral two is written `λs.λz.(s(sz))`, whereas with de Bruijn indices it is written `λλ(1(10))`. In the latter variables names do not appear at point of binding, and instead each variable is replaced by a count (starting at zero) of how many binders outward one must step over to find the one that binds this variable. With extrinsic typing, one first gives a syntax of pre-terms and then gives rules assigning types to terms, while with intrinsic typing the syntax of terms and the type rules are defined together. Reynolds [8] introduced the names intrinsic and extrinsic; the distinction between the two is sometimes referred to as Curry-style (terms exist prior to types) and Church-style (terms make sense only with their types). A textbook development in Agda of both the named-variable/extrinsic and the de Bruijn/intrinsic style can be found in PLFA.

The calculus in ACCL is called *explicit substitution*, or $\lambda\sigma$. Our calculus is called *explicit weakening*, or `λ↑`. In ACCL, substitutions are constructed with four operations (id, shift, cons, compose) and substitution is an explicit operator on terms. Here, substitutions are constructed with three operations (id, weaken, cons, where weaken is a special case combining shift with composition) and weakening is an explicit operator on terms, while substitution and composition become meta operations. For ACCL it is important that substitution is explicit, as this is key in using the calculus to design efficient implementations. Conversely, for us it is important that only weakening is explicit and that substitution and composition are meta operations, as this design supports the proof assistant in automatically simplifying terms—so that equations which were previously difficult to prove become trivial.

The remainder of this paper develops the new formulation as a literate Agda script. Every line of Agda code is included in this paper, and the source is provided as an artifact. Since the paper is a literate Agda script, when you see code in colour that means it has been type-checked by the Agda system, providing assurance that it is correct. The paper is intended to be accessible to anyone with a passing knowledge of proof assistants. Additional detail on how to formalise proofs in Agda can be found in PLFA [11, 7].

2 Formal development

2.1 Module and imports

We begin with bookkeeping: an option pragma that enables rewriting, the module header, and imports from the Agda standard library. The first two imports are required by the pragma, and the others import equality and related operations. Agda supports mixfix syntax, so _≡_ names infix equality.

```
{-# OPTIONS -rewriting #-}
module Weaken where
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; congtiny 2)
open Eq.≡-Reasoning using (begin_; step-≡-|; _∎)
```

(As of agda-stdlib-2.1, one imports `step-≡-|` to define the operator _≡⟨⟩_ used to display chains of equalities.)

2.2 Operator priorities

We declare in advance binding priorities for infix, prefix, and postfix operators. A higher priority indicates tighter binding, and letters `l` or `r` indicate left or right associativity.

```
infix  4 _⊢_ _⊨_
infixl 5 _▷_ _⨟_
infix  5 ƛ_
infixl 7 _·_
infixr 7 _⇒_
infix  8 _↑
```

2.3 Types

A type is either the natural number type (`N) or a function type (_⇒_).

```
data Type : Set where
  ‘N   : Type
  _⇒_ : Type → Type → Type
```

We let A, B, C range over types.

```
variable
  A B C : Type
```

Agda universally quantifies over any free variables named in variable declarations
Here is a function from naturals to naturals.

```
_ : Type
_ = ‘N ⇒ ‘N
```

In Agda, one may use _ as a dummy name that is convenient for examples.

2.4 Contexts

A context is a list of types. The type corresponding to de Bruijn index zero appears at the right end of the list. The empty context is written ∅, and _▷_ adds a type to the end of the list.

```
data Con : Set where
  ∅   : Con
  _▷_ : Con → Type → Con
```

We let Γ, Δ, Θ, Ξ range over environments.

```
variable
  Γ Δ Θ Ξ : Con
```

Here is an environment in which de Bruijn index zero has type natural, and de Bruijn index one is a function from naturals to naturals.

```
_ : Con
_ = ∅ ▷ (‘N ⇒ ‘N) ▷ ‘N
```

2.5 Terms

We write Γ `⊢ A` for the type of terms in context Γ with type `A`. A term is either de Bruijn variable zero (●), the weakening of a term (`M ↑`), a lambda abstraction (`ƛ N`), an application (`L · M`), the number `zero`, the successor of a number `suc M`. (We omit case expressions and recursion, as they add nothing to the exposition.) Any line beginning with two dashes is a comment What is typeset here as a rule is a line of dashes in the source. We take advantage of this to make our term declarations closely resemble the corresponding type rules.

```
data _⊢_ : Con → Type → Set where

  ● :
      ----------------
      Γ ▷ A ⊢ A

  _↑ :
      (M : Γ ⊢ B)
    → ----------------
      Γ ▷ A ⊢ B

  ƛ_ :
      (N : Γ ▷ A ⊢ B)
    → ----------------
      Γ ⊢ A ⇒ B

  _·_ :
      (L : Γ ⊢ A ⇒ B)
      (M : Γ ⊢ A)
    → ----------------
      Γ ⊢ B

  zero :
```

$$\frac{}{\Gamma \vdash \text{`}N}$$

suc :

$$\rightarrow \frac{(M : \Gamma \vdash \text{`}N)}{\Gamma \vdash \text{`}N}$$

We let `L`, `M`, `N`, `P`, `Q` range over terms.

```
variable
  L M N P Q : Γ ⊢ A
```

Here is the increment function for naturals.

```
inc : ∅ ⊢ ‘N ⇒ ‘N
inc = ƛ (suc ●)
```

Here is the term for the Church numeral two.

```
two : ∅ ⊢ (A ⇒ A) ⇒ A ⇒ A
two = ƛ (ƛ (● ↑ · (● ↑ · ●)))
```

Here ● corresponds to de Bruijn index zero, and ● ↑ corresponds to de Bruijn index one.

Crucially, weakening can be applied to any term, not just a variable. The following two open terms are equivalent.

```
M₀ M₁ : ∅ ▷ (A ⇒ B ⇒ C) ▷ A ▷ B ⊢ C
M₀ = ● ↑ ↑ · ● ↑ · ●
M₁ = (● ↑ · ●) ↑ · ●
```

Here ● ↑ ↑ corresponds to de Bruijn index two.

2.6 Substitutions

We write $\Gamma \models \Delta$ for the type of a substitution that replaces variables in environment Δ by terms in environment Γ. A substitution is either the identity substitution `id`, the weakening of a substitution σ ↑, or the cons of a substitution and a term σ ▷ `P`. We take advantage of overloading to permit weakening on both terms `M` ↑ and substitutions σ ↑, and to permit cons on both environments Γ ▷ `A` and substitutions σ ▷ `P`.

```
data _⊨_ : Con → Con → Set where
```

id :

$$\frac{}{\Delta \models \Delta}$$

_↑ :

$$\rightarrow \frac{(\sigma : \Gamma \models \Delta)}{\Gamma \rhd A \models \Delta}$$

_▷_ :

$$\begin{array}{l} (\sigma : \Gamma \models \Delta) \\ (M : \Gamma \vdash A) \\ \rightarrow \overline{} \\ \quad \Gamma \models \Delta \triangleright A \end{array}$$

We let σ, τ, υ range over substitutions.

```
variable
  σ τ υ : Γ ⊨ Δ
```

We can think of a substitution $\Gamma \models \Delta$ built with repeated uses of cons as a list of terms in context Γ, with one term for each type in Δ. For example, here is a substitution that replaces de Bruijn index zero by the number zero, and de Bruijn index one by the increment function on naturals. Each term in the substitution is a closed term, as indicated by the source of the substitution being the empty environment. Here `id` has type $\varnothing \models \varnothing$, as there are no other free variables.

```
_ : ∅ ⊨ ∅ ▷ (‘N ⇒ ‘N) ▷ ‘N
_ = id ▷ inc ▷ zero
```

Here is a substitution that replaces de Bruijn index one by the weakening of the increment function, and leaves de Bruijn index zero unchanged.

```
_ : ∅ ▷ ‘N ⊨ ∅ ▷ ‘N ⇒ ‘N ▷ ‘N
_ = (id ▷ inc) ↑ ▷ ●
```

If we omitted the weakening, the substitution would not be well typed. This is an advantage of the intrinsic approach: correctly maintaining de Bruijn indices is notoriously tricky, but with intrinsic typing getting a de Bruijn index wrong usually leads to a type error.

Finally, here is a substitution that flips two variables, making the outermost innermost and vice versa.

```
_ : ∅ ▷ A ▷ B ⊨ ∅ ▷ B ▷ A
_ = id ↑ ↑ ▷ ● ▷ ● ↑
```

Here `id ↑ ↑ : ∅ ▷ A ▷ B ⊨ ∅`, as required by the type rules for cons. Again, the types keep us straight. If we accidentally add or drop an `↑` the term will become ill-typed.

Often, we will need to know that a substitution is a cons, but will want to refer to the whole substitution. For this purpose, it is convenient to use a pattern declaration to declare △ as a shorthand for `_ ▷ _`.

```
pattern △ = _ ▷ _
```

We can then pattern match against `(σ @ △)`, which binds σ to a substitution, but only if it is in the form of a cons.

2.7 Instantiation

We write `M [ σ ]` to instantiate term `M` with substitution σ. Instantiation is contravariant: substitution `σ : Γ ⊨ Δ` takes a term `M : Δ ⊢ A` into a term `M [ σ ] : Γ ⊢ A`.

```
_[_] :
  (M : Δ ⊢ A)
```

```
   (σ : Γ ⊨ Δ)
   → ____________
      Γ ⊢ A
M        [ id ]      = M                        -- (1)
M        [ σ ↑ ]     = (M [ σ ]) ↑              -- (2)
●        [ σ ▷ P ]   = P                        -- (3)
(M ↑)    [ σ ▷ P ]   = M [ σ ]                  -- (4)
(ƛ N)    [ σ @ △ ]   = ƛ (N [ σ ↑ ▷ ● ])        -- (5)
(L · M)  [ σ @ △ ]   = L [ σ ] · M [ σ ]        -- (6)
zero     [ σ @ △ ]   = zero                     -- (7)
(suc M)  [ σ @ △ ]   = suc (M [ σ ])            -- (8)
```

Instantiation is defined by case analysis. It is crucial that we first perform case analysis on the substitution, since if it is identity or weakening we can compute the result easily without considering the structure of a term—that is the whole point of representing identity and weakening explicitly! Only if the substitution is a cons do we need to perform a case analysis on the term.

Thus, we first do case analysis on the substitution. (1) For identity the definition is obvious. (2) Weakening of a substitution becomes weakening of a term—this is why we defined explicit weakening in the first place. If the substitution is a cons, we then do a case analysis on the term. (3) If the term is de Bruijn variable zero we take the term from the cons. (4) If the term is a weakening we drop the term from the cons and apply recursively. Otherwise, we push the substitution into the term. (5) If the term is a lambda abstraction, the substitution σ becomes $\sigma \uparrow \triangleright \bullet$ as it is pushed under the lambda. Bound variable zero is left unchanged, while variables one and up now map to the terms bound by σ, weakened to account for the newly intervening lambda binder. Our notation neatly encapsulates all renumbering of de Bruijn indexes, which is usually considered tricky. (6–8) Otherwise, the substitution is unchanged and applied recursively to each part. This completes the definition of instantiation.

Beta reduction is written as follows.

```
(ƛ N) · M  —→  N [ id ▷ M ]
```

Substution `id ▷ M` maps variable zero to `M`, and maps variable one to zero, two to one, and so on, as required since the surrounding lambda binder has been removed. Once again, our notation neatly encodes the tricky renumbering of de Bruijn indexes.

Consider an application of the Church numeral two to the increment function.

```
two · inc
```

Beta reduction yields an application of a substitution, which we compute as follows. Each equality is labelled with its justifying line from the definition.

```
_ : (ƛ (● ↑ · (● ↑ · ●))) [ id ▷ inc ] ≡ ƛ (inc ↑ · (inc ↑ · ●))
_ =
  begin
    (ƛ (● ↑ · (● ↑ · ●))) [ id ▷ inc ]
  ≡⟨⟩ -- (5)
    ƛ ((● ↑ · (● ↑ · ●)) [ (id ▷ inc) ↑ ▷ ● ])
  ≡⟨⟩ -- (6)
    ƛ (((● ↑) [ (id ▷ inc) ↑ ▷ ● ]) · ((● ↑ · ●) [ (id ▷ inc) ↑ ▷ ● ]))
  ≡⟨⟩ -- (4)
    ƛ ((● [ (id ▷ inc) ↑ ]) · ((● ↑ · ●) [ (id ▷ inc) ↑ ▷ ● ]))
  ≡⟨⟩ -- (2)
  ƛ ((● [ id ▷ inc ]) ↑ · ((● ↑ · ●) [ (id ▷ inc) ↑ ▷ ● ]))
≡⟨⟩ - (3)
  ƛ (inc ↑ · ((● ↑ · ●) [ (id ▷ inc) ↑ ▷ ● ]))
≡⟨⟩ -   ...
  ƛ (inc ↑ · (inc ↑ · ●))
∎
```

Since `inc` is a closed term, it must be weakened to appear underneath a lambda, and this is accomplished by the rule that converts σ to $\sigma \uparrow \triangleright \bullet$ when pushing a substitution under a lambda abstraction. The end of the computation adds nothing new, so some details are omitted. Indeed all the detail can be omitted, as Agda can confirm the result simply by normalising both sides.

```
_ : (ƛ (● ↑ · (● ↑ · ●))) [ id ▷ inc ] ≡ ƛ (inc ↑ · (inc ↑ · ●))
_ = refl
```

2.8 Composition

We write $\sigma \fatsemi \tau$ for composition of substitutions. If $\sigma : \Theta \models \Delta$ and $\tau : \Gamma \models \Theta$ then $(\sigma \fatsemi \tau) : \Gamma \Rrightarrow \Delta$.

```
_⨾_ :
    (σ : Θ ⊨ Δ)
    (τ : Γ ⊨ Θ)
  → ________________________
      Γ ⊨ Δ
σ         ⨾ id        = σ                   - (1)
σ         ⨾ (τ ↑)     = (σ ⨾ τ) ↑           - (2)
id        ⨾ (τ @ △)   = τ                   - (3)
(σ ↑)     ⨾ (τ ▷ Q)   = σ ⨾ τ               - (4)
(σ ▷ P)   ⨾ (τ @ △)   = (σ ⨾ τ) ▷ (P [ τ ]) - (5)
```

The case analysis for instantiation goes right-to-left: first we analyse the substitution, and only if it is a cons do we analyse the term. Hence, composition is also defined right-to-left: first we analyse the right substitution τ, and only if it is a cons do we analyse the left substitution σ. Each of the equations should be familiar by now.

2.9 Composition and instantiation

A key result relates composition and instantiation.

```
[][] :
  (M : Δ ⊢ A)
  (σ : Θ ⊨ Δ)
  (τ : Γ ⊨ Θ)
  → ________________________
      M [ σ ] [ τ ] ≡ M [ σ ⨾ τ ]
```

The only tricky step is to recognise that case analysis of the arguments must proceed right-to-left, to match the definitions of application and composition. Hence, first we perform case analysis on τ, and only if τ is a cons do we perform case analysis on σ, and only if both are conses do we perform case analysis on `M`. Below we use the operator `cong`, short for congruence. If eq proves `x ≡ y`, then `cong f eq` proves `f x ≡ f y`. Similarly for `congtiny 2`.

```
[][] M        σ          id          = refl                                   -- (1)
[][] M        σ          (τ ↑)       = cong _↑ ([][] M σ τ)                   -- (2)
[][] M        id         (τ @ △)     = refl                                   -- (3)
[][] M        (σ ↑)      (τ ▷ Q)     = [][] M σ τ                             -- (4)
[][] •        (σ ▷ P)    (τ @ △)     = refl                                   -- (5)
[][] (M ↑)    (σ ▷ P)    (τ @ △)     = [][] M σ τ                             -- (6)
[][] (ƛ N)    (σ @ △)    (τ @ △)     = cong ƛ_ ([][] N (σ ↑ ▷ •) (τ ↑ ▷ •))   -- (7)
[][] (L · M)  (σ @ △)    (τ @ △)     = congtiny 2 _·_ ([][] L σ τ) ([][] M σ τ) -- (8)
[][] zero     (σ @ △)    (τ @ △)     = refl                                   -- (9)
[][] (suc M)  (σ @ △)    (τ @ △)     = cong suc ([][] M σ τ)                  -- (10)
```

For instance, for (2) the two sides simplify to

`M [ σ ] [ τ ] ↑  ≡  M [ σ ⨾ τ ] ↑`

and the equation follows by an application of the induction hypothesis. Similarly, for (8) the two sides simplify to

`(L [ σ ] [ τ ]) · (M [ σ ] [ τ ]) ≡ (L [ σ ⨾ τ ]) · (M [ σ ⨾ τ ])`

and the equation follows by two applications of the induction hypothesis. For (7), we push substitutions underneath a lambda, giving

`ƛ (N [ σ ↑ ▷ • ] [ τ ↑ ▷ • ]) ≡ N [ (σ ↑ ▷ •) ⨾ (τ ↑ ▷ •) ] ≡ N [ (σ ⨾ τ) ↑ ▷ • ]`

where the first equivalence follows by the induction hypothesis, but applied to the substitutions `σ ↑ ▷ •` and `τ ↑ ▷ •`. It is fine under structural induction for the substitutions to get larger so long as the term is getting smaller. The second equivalence follows by straightforward computation.

Having proved the lemma, we can now instruct Agda to apply it as a left-to-right rewrite whenever possible when simplifying a term. This will play a key role in reducing equations of interest to a triviality.

```
{-# REWRITE [][] #-}
```

2.10 Composition has a left identity.

Composition has `id` as a right identity by definition. It is easy to show that it is also a left identity.

```
left-id :
  (τ : Γ ⊨ Δ)
  → ─────────────────
      id ⨾ τ ≡ τ
left-id id        = refl                  -- (1)
left-id (τ ↑)     = cong _↑ (left-id τ)   -- (2)
left-id (τ ▷ Q)   = refl                  -- (3)
```

Obviously, the case analysis is on τ. (1, 3): Both sides simplify to the same term. (2): The two sides simplify to

`(id ⨾ τ) ↑  ≡  τ ↑`

and the result follows by the induction hypothesis.

We direct Agda to apply left identity as a rewrite.

```
{-# REWRITE left-id #-}
```

2.11 Composition is associative

We can also show that composition is associative.

```
assoc :
  (σ : Θ ⊨ Δ)
  (τ : Ξ ⊨ Θ)
  (υ : Γ ⊨ Ξ)
  → ────────────────────────
      (σ ⨾ τ) ⨾ υ ≡ σ ⨾ (τ ⨾ υ)
assoc σ       τ       id      = refl                                            -- (1)
assoc σ       τ       (υ ↑)   = cong _↑ (assoc σ τ υ)                           -- (2)
assoc σ       id      (υ ▷ R) = refl                                            -- (3)
assoc σ       (τ ↑)   (υ ▷ R) = assoc σ τ υ                                     -- (4)
assoc id      (τ ▷ Q) (υ ▷ R) = refl                                            -- (5)
assoc (σ ↑)   (τ ▷ Q) (υ ▷ R) = assoc σ τ (υ ▷ R)                               -- (6)
assoc (σ ▷ P) (τ ▷ Q) (υ ▷ R) = congtiny 2 _▷_ (assoc σ (τ ▷ Q) (υ ▷ R)) refl -- (7)
```

Again, the only tricky step is to recognise that case analysis on the arguments must proceed right-to-left. First we perform case analysis on υ, and only if υ is a cons do we perform case analysis on τ, and only if both are conses do we perform case analysis on σ. For instance, in (6) the two sides simplify to

```
((σ ⨾ τ) ⨾ (υ ▷ R)) ≡ (σ ⨾ (τ ⨾ (υ ▷ R)))
```

and the equation follows by the induction hypothesis on σ, τ, and `υ ▷ R`.

We direct Agda to apply associativity as a rewrite.

```
{-# REWRITE assoc #-}
```

3 Applications

3.1 Special cases of substitution

We define three special cases of substitution.

Substitute for the last variable in the environment (de Bruijn index zero).

```
_[_]₀ :
  (N : Γ ▷ A ⊢ B)
  (M : Γ ⊢ A)
  → ────────────────────────
      Γ ⊢ B
N [ M ]₀ = N [ id ▷ M ]
```

This is exactly what we need for beta reduction.

Substitute for the last but one variable in the environment (de Bruijn index one).

```
_[_]₁ :
  (N : Γ ▷ A ▷ B ⊢ C)
  (M : Γ ⊢ A)
  → ────────────────────────
      Γ ▷ B ⊢ C
N [ M ]₁ = N [ (id ▷ M) ↑ ▷ • ]
```

3.2 An example

The first equation given in the introduction holds trivially.

```
introduction : (N ↑) [ M ]₀ ≡ N
introduction = refl
```

It is straightforward to see that the left-hand side simplifies to the right-hand side. Recall that this took nearly a hundred lines to prove in the formulation that we used previously!

3.3 A challenging exercise

The following exercise appears in PLFA. It is marked “stretch” meaning it is intended to be challenging.

```
double-subst : N [ M ]₁ [ L ]₀ ≡ N [ L ↑ ]₀ [ M ]₀
```

PLFA uses a very different formulation of substitution than the one given here, and under that formulation the exercise appears quite challenging—indeed, so far as I know, no one has solved it!

However, with the formulation given here, the exercise becomes trivial.

```
double-subst = refl
```

Both sides simplify by an automatic rewrite with `lemma-⨾` and then normalising the compositions yields identical terms.

3.4 A second challenge

The following result is the culmination of Chapter Substitution of PLFA.

```
commute-subst : N [ M ]₀ [ L ]₀ ≡ N [ L ]₁ [ M [ L ]₀ ]₀
```

In effect, the entire chapter is devoted to proving it. A theory similar to that of ACCL is developed at length, requiring a few hundred lines of Agda. Even once the theory is developed, the key lemma, `subst-commute`, requires a chain of thirteen equations to prove, eleven of which required justification.

However, with the formulation given here, the result becomes trivial.

```
commute-subst = refl
```

Both sides simplify by an automatic rewrite with `lemma-⨾` and then normalising the compositions yields identical terms.

4 Conclusion

A drawback of this technique is that it distinguishes terms that in traditional notation must be equivalent, such as the terms `● ↑ ↑ · ● ↑ · ●` and `(● ↑ · ●) ↑ · ●` mentioned previously. As a result, identifying when terms are equivalent may become more difficult. On the other hand, if equivalence is important we are often concerned with normal forms (such as β and η normal forms) and in that case pushing `↑` to the inside as part of normalisation would cause the second term to normalise to the first, removing the problem of determining equivalence.

Further experience is required. Is explicit weakening useful in practice? Time will tell.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien & J.-J. Levy (1989): *Explicit substitutions*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, pp. 31–46, doi:10.1145/96709.96712. Available at `https://dl.acm.org/doi/10.1145/96709.96712`.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien & Jean-Jacques Levy (1991): *Explicit Substitutions*. *Journal of Functional Programming* 1, pp. 375–416, doi:10.1017/S0956796800000186.

[3] Thorsten Altenkirch & Bernhard Reus (1999): *Monadic Presentations of Lambda Terms Using Generalized Inductive Types*. In Jörg Flum & Mario Rodriguez-Artalejo, editors: *Computer Science Logic*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 453–468, doi:10.1007/3-540-48168-0_32.

[4] René David & Bruno Guillaume (2001): *A λ-calculus with explicit weakening and explicit substitution*. *Mathematical Structures in Computer Science* 11(1), pp. 169–206, doi:10.1017/S0960129500003224. Available at `https://www.cambridge.org/core/journals/mathematical-structures-in-computer-science/article/calculus-with-explicit-weakening-and-explicit-substitution/2A6EB436BB194308795433464909C770`. Publisher: Cambridge University Press.

[5] Dimitri Hendriks & Vincent van Oostrom (2003): $\reflectbox{$\lambda$}$. In Franz Baader, editor: *Automated Deduction – CADE-19*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 136–150, doi:10.1007/978-3-540-45085-6_11.

[6] Delia Kesner (2009): *A Theory of Explicit Substitutions with Safe and Full Composition*. *Logical Methods in Computer Science* 5(3), p. 1, doi:10.2168/LMCS-5(3:1)2009. Available at `https://lmcs.episciences.org/816`.

[7] Wen Kokke, Jeremy G. Siek & Philip Wadler (2020): *Programming language foundations in Agda*. *Science of Computer Programming* 194, p. 102440, doi:10.1016/j.scico.2020.102440. Available at `https://www.sciencedirect.com/science/article/pii/S0167642320300502`.

[8] John C. Reynolds (2000): *The Meaning of Types From Intrinsic to Extrinsic Semantics*. *BRICS Report Series* (32), doi:10.7146/brics.v7i32.20167. Available at `https://tidsskrift.dk/brics/article/view/20167`. Number: 32.

[9] Kristoffer H. Rose, Roel Bloo & Frédéric Lang (2012): *On Explicit Substitution with Names*. *Journal of Automated Reasoning* 49(2), pp. 275–300, doi:10.1007/s10817-011-9222-5.

[10] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions*. In Christian Urban & Xingyuan Zhang, editors: *Interactive Theorem Proving*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[11] Philip Wadler, Wen Kokke & Jeremy G Siek (2018): *Programming Language Foundations in Agda*. doi:10.1007/978-3-030-03044-5_5, Available at `https://plfa.inf.ed.ac.uk/`.