Substitution without copy and paste

Thorsten Altenkirch ✉
University of Nottingham, Nottingham, United Kingdom

Nathaniel Burke ✉
Imperial College London, London, United Kingdom

Philip Wadler ✉
University of Edinburgh, Edinburgh, United Kingdom

Abstract

When defining substitution recursively for a language with binders like the simply typed λ-calculus, we need to define substitution and renaming separately. When we want to verify the categorical properties of this calculus, we end up repeating the same argument many times. In this paper we present a lightweight method that avoids this repetition and is implemented in Agda.

We use our setup to also show that the recursive definition of substitution gives rise to a simply typed category with families (CwF) and indeed that it is isomorphic to the initial simply typed CwF.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Substitution, Metatheory, Agda

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

> Some half dozen persons have written technically on combinatory logic, and most of these, including ourselves, have published something erroneous. [9]

The first author was writing lecture notes for an introduction to category theory for functional programmers. A nice example of a category is the category of simply typed λ-terms and substitutions; hence it seemed a good idea to give the definition and ask the students to prove the category laws. When writing the answer, they realised that it is not as easy as they thought, and to make sure that there were no mistakes, they started to formalize the problem in Agda. The main setback was that the same proofs got repeated many times. If there is one guideline of good software engineering then it is **Do not write code by copy and paste** and this applies even more so to formal proofs.

This paper is the result of the effort to refactor the proof. We think that the method used is interesting also for other problems. In particular the current construction can be seen as a warmup for the recursive definition of substitution for dependent type theory which may have interesting applications for the coherence problem, i.e. interpreting dependent types in higher categories.

1.1 In a nutshell

When working with substitution for a calculus with binders, we find that you have to differentiate between renamings (Δ ⊨v Γ) where variables are substituted only for variables (Γ ∋ A) and proper substitutions (Δ ⊨ Γ) where variables are replaced with terms (Γ ⊢ A). This results in having to define several similar operations

```
_v[_]v : Γ ∋ A → Δ ⊨v Γ → Δ ∋ A
_v[_]  : Γ ∋ A → Δ ⊨ Γ  → Δ ⊢ A
```

```
_[_]v  : Γ ⊢ A → Δ ⊨v Γ → Δ ⊢ A
_[_]   : Γ ⊢ A → Δ ⊨ Γ  → Δ ⊢ A
```

And indeed the operations on terms depend on the operations on variables. This duplication gets worse when we prove properties of substitution, such as the functor law:

```
x [ xs ∘ ys ] ≡ x [ xs ] [ ys ]
```

Since all components x, xs, ys can be either variables/renamings or terms/substitutions, we seemingly need to prove eight possibilities (with the repetition extending also to the intermediary lemmas). Our solution is to introduce a type of sorts with V : Sort for variables/renamings and T : Sort for terms substitutions, leading to a single substitution operation

```
_[_] : Γ ⊢[ q ] A → Δ ⊨[ r ] Γ → Δ ⊢[ q ⊔ r ] A
```

where q, r : Sort and q ⊔ r is the least upper bound in the lattice of sorts (V ⊑ T). With this, we only need to prove one variant of the functor law, relying on the fact that _⊔_ is associative. We manage to convince Agda's termination checker that V is structurally smaller than T (see section 3) and, indeed, our highly mutually recursive proof relying on this is accepted by Agda.

We also relate the recursive definition of substitution to a specification using a quotient-inductive-inductive type (QIIT) (a mutual inductive type with equations) where substitution is a term former (i.e. explicit substitutions). Specifically, our specification is such that the substitution laws correspond to the equations of a simply typed category with families (CwF) (a variant of a category with families where the types do not depend on a context). We show that our recursive definition of substitution leads to a simply typed CwF which is isomorphic to the specified initial one. This can be viewed as a normalisation result where the usual λ-terms without explicit substitutions are the *substitution normal forms*.

1.2 Related work

[10] introduces his eponymous indices and also the notion of simultaneous substitution. We are here using a typed version of de Bruijn indices, e.g. see [6] where the problem of showing termination of a simple definition of substitution (for the untyped λ-calculus) is addressed using a well-founded recursion. The present approach seems to be simpler and scales better, avoiding well-founded recursion. Andreas Abel used a very similar technique to ours in his unpublished Agda proof [1] for untyped λ-terms when implementing [6].

The monadic approach has been further investigated in [13]. The structure of the proofs is explained in [3] from a monadic perspective. Indeed this example is one of the motivations for relative monads [7].

In the monadic approach, we represent substitutions as functions, however it is not clear how to extend this to dependent types without "very dependent" types.

There are a number of publications on formalising substitution laws. Just to mention a few recent ones: [17] develops a Coq library which automatically derives substitution lemmas, but the proofs are repeated for renamings and substitutions. Their equational theory is similar to the simply typed CwFs we are using in section 5. [15] is also using Agda, but extrinsically (i.e. separating preterms and typed syntax). Here the approach from [3] is used to factor the construction using *kits*. [16] instead uses intrinsic syntax, but with renamings and substitutions defined separately, and relevant substitution lemmas repeated for all required combinations.

1.3 Using Agda

For the technical details of Agda we refer to the online documentation [18]. We only use plain Agda, inductive definitions and structurally recursive programs and proofs. Termination is checked by Agda's termination checker [2] which uses a lexical combination of structural descent that is inferred by the termination checker by investigating all possible recursive paths. We will define mutually recursive proofs which heavily rely on each other.

The only recent feature we use, albeit sparingly, is the possibility to turn propositional equations into rewriting rules (i.e. definitional equalities). This makes the statement of some theorems more readable because we can avoid using subst, but it is not essential.

We extensively use variable declarations to introduce implicit quantification (we summarize the variable conventions in passing in the text). We also use ∀-prefix so we can elide types of function parameters where they can be inferred, i.e. instead of {Γ : Con} → .. we just write ∀ {Γ} → ... Implicit variables, which are indicated by using {..} instead of (..) in dependent function types, can be instantiated using the syntax a {x = b}.

Agda syntax is very flexible, allowing mixfix syntax declarations using _ to indicate where the parameters go. In the proofs, we use the Agda standard library's definitions for equational derivations, which exploit this flexibility.

The source of this document contains the actual Agda code, i.e. it is a literate Agda file. Different chapters are in different modules to avoid name clashes, e.g. preliminary definitions from section 2 are redefined later.

2 The naive approach

Let us first review the naive approach which leads to the copy-and-paste proof. We define types (A, B, C) and contexts (Γ, Δ, Θ):

```
data Ty : Set where
  o : Ty
  _⇒_ : Ty → Ty → Ty
data Con : Set where
  • : Con
  _▷_ : Con → Ty → Con
```

Next we introduce intrinsically typed de Bruijn variables (i, j, k) and λ-terms (t, u, v) :

```
data _∋_ : Con → Ty → Set where
  zero : Γ ▷ A ∋ A
  suc  : Γ ∋ A → (B : Ty) → Γ ▷ B ∋ A
data _⊢_ : Con → Ty → Set where
  `_   : Γ ∋ A → Γ ⊢ A
  _·_  : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
  λ_   : Γ ▷ A ⊢ B → Γ ⊢ A ⇒ B
```

Here the constructor `_ corresponds to *variables are λ-terms*. We write applications as t · u. Since we use de Bruijn variables, lambda abstraction λ_ doesn't bind a name explicitly (instead, variables count the number of binders between them and their actual binding site). We also define substitutions as sequences of terms:

```
data _⊨_ : Con → Con → Set where
  ε : Γ ⊨ •
  _,_ : Γ ⊨ Δ → Γ ⊢ A → Γ ⊨ Δ ▷ A
```

Now to define the categorical structure (_∘_, id) we first need to define substitution for terms and variables:

```
_v[_] : Γ ∋ A → Δ ⊨ Γ → Δ ⊢ A
zero v[ ts , t ]      = t
(suc i _) v[ ts , t ] = i v[ ts ]
_[_] : Γ ⊢ A → Δ ⊨ Γ → Δ ⊢ A
(` i) [ ts ]     = i v[ ts ]
(t · u) [ ts ] = (t [ ts ]) · (u [ ts ])


(λ t) [ ts ] = λ ?
```

As usual, we encounter a problem with the case for binders λ_. We are given a substitution ts : Δ ⊨ Γ but the body t lives in the extended context t : Γ , A ⊢ B. We need to exploit the fact that context extension _ ▷ _ is functorial:

```
_ ↑ _ : Γ ⊨ Δ → (A : Ty) → Γ ▷ A ⊨ Δ ▷ A
```

Using _ ↑ _ we can complete _[_]

```
(λ t) [ ts ] = λ (t [ ts ↑ _ ])
```

However, now we have to define _ ↑ _. This is easy (isn't it?) but we need weakening on substitutions:

```
_⁺_ : Γ ⊨ Δ → (A : Ty) → Γ ▷ A ⊨ Δ
```

And now we can define _ ↑ _:

```
ts ↑ A = ts ⁺ A , ` zero
```

but we need to define _⁺_, which is nothing but a fold of weakening of terms

```
suc-tm : Γ ⊢ B → (A : Ty) → Γ ▷ A ⊢ B
ε       ⁺ A = ε
(ts , t) ⁺ A = ts ⁺ A , suc-tm t A
```

But how can we define suc-tm when we only have weakening for variables? If we already had identity id : Γ ⊨ Γ and substitution we could write:

```
suc-tm t A = t [ id ⁺ A ]
```

but this is certainly not structurally recursive (and hence rejected by Agda's termination checker).

Actually, we realize that id is a renaming, i.e. it is a substitution only containing variables, and we can easily define ⁺v for renamings. This leads to a structurally recursive definition, but we have to repeat the definition of substitutions for renamings.

```
data _ ⊨v _ : Con → Con → Set where
  ε : Γ ⊨v •
  _,_ : Γ ⊨v Δ → Γ ∋ A → Γ ⊨v Δ ▷ A
⁺v : Γ ⊨v Δ → (A : Ty) → Γ ▷ A ⊨v Δ
```

```
ε        ⁺v A      = ε
(is , i) ⁺v A      = is ⁺v A , suc i A
_↑v_ : Γ ⊨v Δ → (A : Ty) → Γ ▷ A ⊨v Δ ▷ A
is ↑v A = is ⁺v A , zero
_v[_]v : Γ ∋ A → Δ ⊨v Γ → Δ ∋ A
zero  v[ is , i ]v      = i
(suc i _) v[ is , j ]v  = i v[ is ]v
_[_]v : Γ ⊢ A → Δ ⊨v Γ → Δ ⊢ A
(` i)  [ is ]v     = ` (i v[ is ]v)
(t · u) [ is ]v = (t [ is ]v) · (u [ is ]v)
(λ t) [ is ]v      = λ (t [ is ↑v _ ]v)
idv : Γ ⊨v Γ
idv {Γ = •} =        ε
idv {Γ = Γ ▷ A} = idv ↑v A
suc-tm t A = t [ idv ⁺v A ]v
```

This may not sound too bad: to obtain structural termination we just have to duplicate a few definitions, but it gets even worse when proving the laws. For example, to prove associativity, we first need to prove functoriality of substitution:

```
[∘] : t [ us ∘ vs ] ≡ t [ us ] [ vs ]
```

Since t, us, vs can be variables/renamings or terms/substitutions, there are in principle eight combinations (though it turns out that four is enough). Each time, we must to prove a number of lemmas again in a different setting.

In the rest of the paper we describe a technique for factoring these definitions and the proofs, only relying on the Agda termination checker to validate that the recursion is structurally terminating.

3 Factorising with sorts

Our main idea is to turn the distinction between variables and terms into a parameter. The first approximation is to define a type Sort (q, r, s) :

```
data Sort : Set where
  V T : Sort
```

but this is not exactly what we want because we want Agda to know that the sort of variables V is *smaller* than the sort of terms T (following intuition that variable weakening is trivial, but to weaken a term we must construct a renaming). Agda's termination checker only knows about the structural orderings. With the following definition, we can make V structurally smaller than T>V V isV, while maintaining that Sort has only two elements.

```
data Sort : Set
data IsV : Sort → Set
data Sort where
  V : Sort
  T>V : (s : Sort) → IsV s → Sort
data IsV where
  isV : IsV V
```

Here the predicate isV only holds for V. This particular encoding makes use of Agda's support for inductive-inductive datatypes (IITs), but merely a pair of a natural number n and a proof n ⩽ 1 is sufficient:

```
Sort : Set
Sort = Σ ℕ (_⩽ 1)
```

We can now define T = T>V V isV : Sort but, even better, we can tell Agda that this is a derived pattern

```
pattern T = T>V V isV
```

This means we can pattern match over Sort just with V and T, while ensuring V is visibly (to Agda's termination checker) structurally smaller than T.

We can now define terms and variables in one go (x, y, z):

```
data _⊢[_]_ : Con → Sort → Ty → Set where
  zero : Γ ▷ A ⊢[ V ] A
  suc  : Γ ⊢[ V ] A → (B : Ty) → Γ ▷ B ⊢[ V ] A
  `_   : Γ ⊢[ V ] A → Γ ⊢[ T ] A
  _·_  : Γ ⊢[ T ] A ⇒ B → Γ ⊢[ T ] A → Γ ⊢[ T ] B
  λ_   : Γ ▷ A ⊢[ T ] B → Γ ⊢[ T ] A ⇒ B
```

While almost identical to the previous definition (Γ ⊢[V] A corresponds to Γ ∋ A and Γ ⊢[T] A to Γ ⊢ A) we can now parametrize all definitions and theorems explicitly. As a first step, we can generalize renamings and substitutions (xs, ys, zs):

```
data _⊨[_]_ : Con → Sort → Con → Set where
  ε : Γ ⊨[ q ] •
  _,_ : Γ ⊨[ q ] Δ → Γ ⊢[ q ] A → Γ ⊨[ q ] Δ ▷ A
```

To account for the non-uniform behaviour of substitution and composition (the result is V only if both inputs are V) we define a least upper bound on Sort:

```
_⊔_ : Sort → Sort → Sort
V ⊔ r = r
T ⊔ r = T
```

We also need this order as a relation, for inserting coercions when necessary:

```
data _⊑_ : Sort → Sort → Set where
  rfl : s ⊑ s
  v⊑t : V ⊑ T
```

Yes, this is just boolean algebra. We need a number of laws:

```
⊑t : s ⊑ T
v⊑ : V ⊑ s
⊑q⊔ : q ⊑ (q ⊔ r)
⊑⊔r : r ⊑ (q ⊔ r)
⊔⊔ : q ⊔ (r ⊔ s) ≡ (q ⊔ r) ⊔ s
⊔v : q ⊔ V ≡ q
```

which are easy to prove by case analysis, e.g.

```
⊑t {V} = v⊑t
⊑t {T} = rfl
```

To improve readability we turn the equations (⊔⊔, ⊔v) into rewrite rules: by declaring

```
{-# REWRITE ⊔⊔ ⊔v #-}
```

This introduces new definitional equalities, i.e. `q ⊔ (r ⊔ s) = (q ⊔ r) ⊔ s` and `q ⊔ V = q` are now used by the type checker. [1] The order gives rise to a functor which is witnessed by

```
tm⊑ : q ⊑ s → Γ ⊢[ q ] A → Γ ⊢[ s ] A
tm⊑ rfl x = x
tm⊑ v⊑t i = ` i
```

Using a parametric version of _↑_

```
_↑_ : Γ ⊨[ q ] Δ → ∀ A → Γ ▷ A ⊨[ q ] Δ ▷ A
```

we are ready to define substitution and renaming in one operation

```
_[_] : Γ ⊢[ q ] A → Δ ⊨[ r ] Γ → Δ ⊢[ q ⊔ r ] A
zero [ xs , x ] =  x
(suc i _) [ xs , x ] = i [ xs ]
(` i) [ xs ]    =  tm⊑ ⊑t (i [ xs ])
(t · u) [ xs ]  =  (t [ xs ]) · (u [ xs ])
(λ t) [ xs ]    =  λ (t [ xs ↑ _ ])
```

We use _⊔_ here to take care of the fact that substitution will only return a variable if both inputs are variables / renamings. We also need to use tm⊑ to take care of the two cases when substituting for a variable.

We can also define id using _↑_:

```
id : Γ ⊨[ V ] Γ
id {Γ = •} =     ε
id {Γ = Γ ▷ A} = id ↑ A
```

To define _↑_, we need parametric versions of zero, suc and suc*. zero is very easy:

```
zero[_] : ∀ q → Γ ▷ A ⊢[ q ] A
zero[ V ] = zero
zero[ T ] = ` zero
```

However, suc is more subtle since the case for T depends on its fold over substitutions (_⁺_):

```
_⁺_ : Γ ⊨[ q ] Δ → (A : Ty) → Γ ▷ A ⊨[ q ] Δ
suc[_] : ∀ q → Γ ⊢[ q ] B → (A : Ty)
```

[1] Effectively, this feature allows a selective use of extensional Type Theory.

```
    → Γ ▷ A ⊢[ q ] B
  suc[ V ] i A  =  suc i A
  suc[ T ] t A  =  t [ id ⁺ A ]

  ε ⁺ A  =  ε
  (xs , x) ⁺ A  =  xs ⁺ A , suc[ _ ] x A
```

And now we define:

```
  xs ↑ A  =  xs ⁺ A , zero[ _ ]
```

3.1 Termination

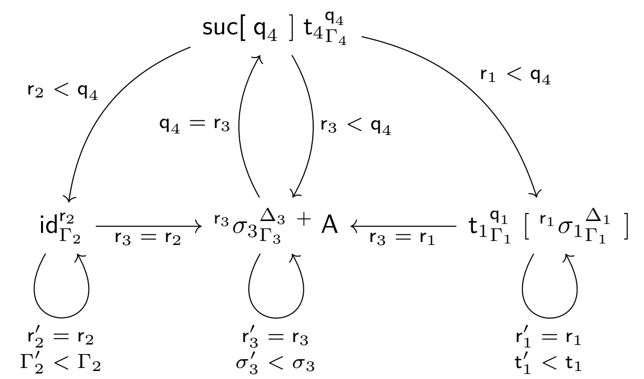
Unfortunately (as of Agda 2.7.0.1), we now hit a termination error.

```
Termination checking failed for the following functions:
  _^_, _[_], id, _⁺_, suc[_]
```

The cause turns out to be id. Termination here hinges on weakening for terms (suc[T] t A) building and applying a renaming (i.e. a sequence of variables, for which weakening is trivial) rather than a full substutution. Note that if id produced Tms[T] Γ Γs, or if we implemented weakening for variables (suc[V] i A) with i [id ⁺ A], our operations would still be type-correct, but would genuinely loop, so perhaps Agda is right to be careful.

Of course, we have specialised weakening for variables, so we now must ask why Agda still doesn't accept our program. The limitation is ultimately a technical one: Agda only looks at the direct arguments to function calls when building the call graph from which it identifies termination order [2]. Because id is not passed a sort, the sort cannot be considered as decreasing in the case of term weakening (suc[T] t A).

Luckily, there is an easy solution here: making id Sort-polymorphic and instantiating with V at the call-sites adds new rows/columns (corresponding to the Sort argument) to the call matrices involving id, enabling the decrease to be tracked and termination to be correctly inferred by Agda. We present the call graph diagramatically (inlining _ ↑ _), in the style of [12].



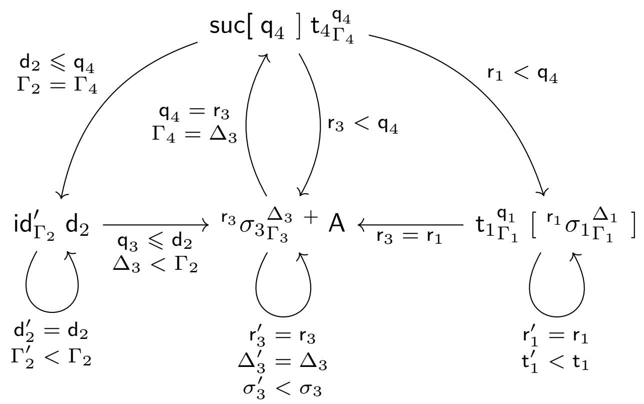
To justify termination formally, we note that along all cycles in the graph, either the Sort strictly decreases in size, or the size of the Sort is preserved and some other argument (the context, substitution or term) gets smaller. We can therefore assign decreasing measures as follows:

Function	Measure
$t_1{}^{q_1}_{\Gamma_1}$ [$^{r_1}\sigma_1{}^{\Delta_1}_{\Gamma_1}$]	$(r_1\ ,\ t_1)$
$\mathsf{id}^{r_2}_{\Gamma_2}$	$(r_2\ ,\ \Gamma_2)$
$^{r_3}\sigma_3{}^{\Delta_3}_{\Gamma_3}$ + A	$(r_3\ ,\ \sigma_3)$
suc[q_4] $t_4{}^{q_4}_{\Gamma_4}$	(q_4)

We now have a working implementation of substitution. In preparation for a similar termination issue we will encounter later though, we note that, perhaps surprisingly, adding a "dummy argument" to id of a completely unrelated type, such as Bool also satisfies Agda. That is, we can write

```
id′ : Bool → Γ ⊨[ V ] Γ
id′ {Γ = •}       d = ε
id′ {Γ = Γ ▷ A} d = id′ d ↑ A
id : Γ ⊨[ V ] Γ
id = id′ true
 {-# INLINE id #-}
```

This result was a little surprising at first, but Agda's implementation reveals answers. It turns out that Agda considers "base constructors" (data constructors taking with arguments) to be structurally smaller-than-or-equal-to all parameters of the caller. This enables Agda to infer true $\leqslant$ T in suc[T] t A and V $\leqslant$ true in id′ {Γ = Γ ▷ A}; we do not get a strict decrease in Sort like before, but the size is at least preserved, and it turns out (making use of some slightly more complicated termination measures) this is enough:



This "dummy argument" approach perhaps is interesting because one could imagine automating this process (i.e. via elaboration or directly inside termination checking). In fact, a PR featuring exactly this extension is currently open on the Agda GitHub repository.

Ultimately the details behind how termination is ensured do not matter here though: both approaches provide effectively the same interface. [2]

Finally, we define composition by folding substitution:

[2] Technically, a Sort-polymorphic id provides a direct way to build identity *substitutions* as well as identity *renamings*, which are useful for implementing single substitutions (< t > = id , t), but we can easily recover this with a monomorphic id by extending tm⊑ to lists of terms (see **??**). For the rest of the paper, we will use id : Γ ⊨[V] Γ without assumptions about how it is implemented.

```
_∘_ : Γ ⊨[ q ] Θ → Δ ⊨[ r ] Γ → Δ ⊨[ q ⊔ r ] Θ
ε ∘ ys        = ε
(xs , x) ∘ ys = (xs ∘ ys) , x [ ys ]
```

4 Proving the laws

We now present a formal proof of the categorical laws, proving each lemma only once while only using structural induction. Indeed the termination isn't completely trivial but is still inferred by the termination checker.

4.1 The right identity law

Let's get the easy case out of the way: the right-identity law (xs ∘ id ≡ xs). It is easy because it doesn't depend on any other categorical equations.

The main lemma is the identity law for the substitution functor:

```
[id] : x [ id ] ≡ x
```

To prove the successor case, we need naturality of suc[q] applied to a variable, which can be shown by simple induction over said variable: [3]

```
⁺-nat[]v : i [ xs ⁺ A ] ≡ suc[ q ] (i [ xs ]) A
⁺-nat[]v {i = zero}    {xs = xs , x} = refl
⁺-nat[]v {i = suc j A} {xs = xs , x} = ⁺-nat[]v {i = j}
```

The identity law is now easily provable by structural induction:

```
[id] {x = zero} = refl
[id] {x = suc i A} =
  i [ id ⁺ A ]
   ≡⟨ ⁺-nat[]v {i = i} ⟩
  suc (i [ id ]) A
   ≡⟨ cong (λ j → suc j A) ([id] {x = i}) ⟩
  suc i A ∎
[id] {x = ` i} =
  cong `_ ([id] {x = i})
[id] {x = t · u} =
  cong₂ _·_ ([id] {x = t}) ([id] {x = u})
[id] {x = λ t} =
  cong λ_ ([id] {x = t})
```

Note that the λ_ case is easy here: we need the law to hold for t : Γ , A ⊢[T] B, but this is still covered by the inductive hypothesis because id {Γ = Γ , A} = id ↑ A.

Note also that is the first time we use Agda's syntax for equational derivations. This is just syntactic sugar for constructing an equational derivation using transitivity and reflexivity, exploiting Agda's flexible syntax. Here e ≡⟨ p ⟩ e' means that p is a proof of e ≡ e'. Later we will also use the special case e ≡⟨⟩ e' which means that e and e' are

[3] We are using the naming conventions introduced in sections 2 and 3, e.g. i : Γ ∋ A.

definitionally equal (this corresponds to e $\equiv\langle$ refl $\rangle$ e' and is just used to make the proof more readable). The proof is terminated with ■ which inserts refl. We also make heavy use of congruence cong f : a ≡ b → f a ≡ f b and a version for binary functions cong_2 g : a ≡ b → c ≡ d → g a c ≡ g b d.

The category law now is a fold of the functor law:

```
∘id : xs ∘ id ≡ xs
∘id {xs = ε} = refl
∘id {xs = xs , x} =
  cong₂ _,_ (∘id {xs = xs}) ([id] {x = x})
```

4.2 The left identity law

We need to prove the left identity law mutually with the second functor law for substitution. This is the main lemma for associativity.

Let's state the functor law but postpone the proof until the next section

```
[∘] : x [ xs ∘ ys ] ≡ x [ xs ] [ ys ]
```

This actually uses the definitional equality [4]

```
⊔⊔ : q ⊔ (r ⊔ s) = (q ⊔ r) ⊔ s
```

because the left hand side has the type

```
Δ ⊢[ q ⊔ (r ⊔ s) ] A
```

while the right hand side has type

```
Δ ⊢[ (q ⊔ r) ⊔ s ] A.
```

Of course, we must also state the left-identity law:

```
id∘ : {xs : Γ ⊨[ r ] Δ}
  → id ∘ xs ≡ xs
```

Similarly to id, Agda will not accept a direct implementation of id∘ as structurally recursive. Unfortunately, adapting the law to deal with a Sort-polymorphic id complicates matters: when xs is a renaming (i.e. at sort V) composed with an identity substition (i.e. at sort T), its sort must be lifted on the RHS (e.g. by extending the tm⊑ functor to lists of terms) to obey _⊔_. Accounting for this lifting is certainly do-able, but in keeping with the single-responsibility principle of software design, we argue it is neater to consider only V-sorted id here and worry about equations involving Sort-coercions later (in **??**).

We therefore use the dummy argument trick, declaring a version of id∘ which takes an unused argument, and implementing our desired left-identity law by instantiating with a suitable base constructor. [5]

---

[4] We rely on Agda's rewrite here. Alternatively we would have to insert a transport using subst.

[5] Alternatively, we could extend sort coercions, tm⊑, to renamings/substitutions. The proofs end up a bit clunkier this way (requiring explicit insertion and removal of these extra coercions).

```
data Dummy : Set where
  ⟨⟩ : Dummy
id∘′ : Dummy → {xs : Γ ⊨[ r ] Δ}
   → id ∘ xs ≡ xs
id∘ = id∘′ ⟨⟩
```

```
 {-# INLINE id∘ #-}
```

To prove it, we need the β-laws for zero[_] and _$^+$_:

```
zero[] : zero[ q ] [ xs , x ] ≡ tm⊑ (⊑⊔r {q = q}) x
⁺∘ : xs ⁺ A ∘ (ys , x) ≡ xs ∘ ys
```

As before we state the laws but prove them later. Now id∘ can be shown easily:

```
id∘′ _ {xs = ε} = refl
id∘′ _ {xs = xs , x} = cong₂ _,_
  (id ⁺ _ ∘ (xs , x)
     ≡⟨ ⁺∘ {xs = id} ⟩
  id ∘ xs
     ≡⟨ id∘ ⟩
  xs ∎)
  refl
```

Now we show the β-laws. zero[] is just a simple case analysis over the sort while $^+\circ$ relies on a corresponding property for substitutions:

```
suc[] : {ys : Γ ⊨[ r ] Δ}
   → (suc[ q ] x _) [ ys , y ] ≡ x [ ys ]
```

The case for q = V is just definitional:

```
suc[] {q = V} = refl
```

while q = T is surprisingly complicated and in particular relies on the functor law [∘].

```
suc[] {q = T} {x = x} {y = y} {ys = ys} =
  (suc[ T ] x _) [ ys , y ]
   ≡⟨⟩
  x [ id ⁺ _ ] [ ys , y ]
   ≡⟨ sym ([∘] {x = x}) ⟩
  x [ (id ⁺ _) ∘ (ys , y) ]
   ≡⟨ cong (λ ρ → x [ ρ ]) ⁺∘ ⟩
  x [ id ∘ ys ]
   ≡⟨ cong (λ ρ → x [ ρ ]) id∘ ⟩
  x [ ys ] ∎
```

Now the β-law $^+\circ$ is just a simple fold. You may note that $^+\circ$ relies on itself indirectly via suc[]. Termination is justified here by the sort decreasing.

4.3 Associativity

We finally get to the proof of the second functor law ([∘] : x [xs ∘ ys] ≡ x [xs] [ys]), the main lemma for associativity. The main obstacle is that for the λ_ case; we need the second functor law for context extension:

```
↑∘ : {xs : Γ ⊨[ r ] Θ} {ys : Δ ⊨[ s ] Γ} {A : Ty}
     → (xs ∘ ys) ↑ A ≡ (xs ↑ A) ∘ (ys ↑ A)
```

To verify the variable case we also need that tm⊑ commutes with substitution, which is easy to prove by case analysis

```
tm[] : tm⊑ ⊑t (x [ xs ]) ≡ (tm⊑ ⊑t x) [ xs ]
```

We are now ready to prove [∘] by structural induction:

```
[∘] {x = zero} {xs = xs , x} = refl
[∘] {x = suc i _} {xs = xs , x} = [∘] {x = i}
[∘] {x = ` x} {xs = xs} {ys = ys} =
  tm⊑ ⊑t (x [ xs ∘ ys ])
    ≡⟨ cong (tm⊑ ⊑t) ([∘] {x = x}) ⟩
  tm⊑ ⊑t (x [ xs ] [ ys ])
    ≡⟨ tm[] {x = x [ xs ]} ⟩
  (tm⊑ ⊑t (x [ xs ])) [ ys ] ∎
[∘] {x = t · u} =
  cong₂ _·_ ([∘] {x = t}) ([∘] {x = u})
[∘] {x = λ t} {xs = xs} {ys = ys} =
  cong λ_ (
    t [ (xs ∘ ys) ↑ _ ]
     ≡⟨ cong (λ zs → t [ zs ]) ↑∘ ⟩
    t [ (xs ↑ _) ∘ (ys ↑ _) ]
     ≡⟨ [∘] {x = t} ⟩
    (t [ xs ↑ _ ]) [ ys ↑ _ ] ∎)
```

From here we prove associativity by a fold:

```
∘∘ : xs ∘ (ys ∘ zs) ≡ (xs ∘ ys) ∘ zs
∘∘ {xs = ε} = refl
∘∘ {xs = xs , x} =
  cong₂ _,_ (∘∘ {xs = xs}) ([∘] {x = x})
```

However, we are not done yet. We still need to prove the second functor law for _↑_ (↑∘). It turns out that this depends on the naturality of weakening:

```
⁺-nat∘ : xs ∘ (ys ⁺ A) ≡ (xs ∘ ys) ⁺ A
```

which unsurprisingly has to be shown by establishing a corresponding property for substitutions:

```
⁺-nat[] : {x : Γ ⊢[ q ] B} {xs : Δ ⊨[ r ] Γ}
  → x [ xs ⁺ A ] ≡ suc[ _ ] (x [ xs ]) A
```

The case q = V is just the naturality for variables which we have already proven:

```
+-nat[] {q = V} {x = i} = +-nat[]v {i = i}
```

The case for q = T is more interesting and relies again on [∘] and ∘id:

```
+-nat[] {q = T} {A = A} {x = x} {xs} =
  x [ xs + A ]
   ≡⟨ cong (λ zs → x [ zs + A ]) (sym ∘id) ⟩
  x [ (xs ∘ id) + A ]
   ≡⟨ cong (λ zs → x [ zs ]) (sym (+−nat∘ {xs = xs})) ⟩
  x [ xs ∘ (id + A) ]
   ≡⟨ [∘] {x = x} ⟩
  x [ xs ] [ id + A ] ■
```

Finally we have all the ingredients to prove the second functor law ↑∘: [6]

```
↑∘ {r = r} {s = s} {xs = xs} {ys = ys} {A = A} =
  (xs ∘ ys) ↑ A
   ≡⟨⟩
  (xs ∘ ys) + A , zero[ r ⊔ s ]
   ≡⟨ cong2 _,_ (sym (+−nat∘ {xs = xs})) refl ⟩
  xs ∘ (ys + A) , zero[ r ⊔ s ]
   ≡⟨ cong2 _,_ refl (tm⊑zero (⊑⊔r {r = s} {q = r})) ⟩
  xs ∘ (ys + A) , tm⊑ (⊑⊔r {q = r}) zero[ s ]
   ≡⟨ cong2 _,_
     (sym (+∘ {xs = xs}))
     (sym (zero[] {q = r} {x = zero[ s ]})) ⟩
  (xs + A) ∘ (ys ↑ A) , zero[ r ] [ ys ↑ A ]
   ≡⟨⟩
  (xs ↑ A) ∘ (ys ↑ A) ■
```

5 Initiality

We can do more than just prove that we have a category. Indeed we can verify the laws of a simply typed category with families (CwF). CwFs are mostly known as models of dependent type theory, but they can be specialised to simple types [8]. We summarize the definition of a simply typed CwF as follows:

- A category of contexts (Con) and substitutions (_ ⊨ _),
- A set of types Ty,
- For every type A a presheaf of terms _ ⊢ A over the category of contexts (i.e. a contravariant functor into the category of sets),
- A terminal object (the empty context) and a context extension operation _ ▷ _ such that Γ ⊨ Δ ▷ A is naturally isomorphic to (Γ ⊨ Δ) × (Γ ⊢ A).

I.e. a simply typed CwF is just a CwF where the presheaf of types is constant. We will give the precise definition in the next section, hence it isn't necessary to be familiar with the categorical terminology to follow the rest of the paper.

[6] Actually we also need that zero commutes with tm⊑: that is for any q⊑r : q ⊑ r we have that tm⊑zero q⊑r : zero[r] ≡ tm⊑ q⊑r zero[q].

We can add further constructors like function types _ ⇒ _. These usually come with a natural isomorphisms, giving rise to β and η laws, but since we are only interested in substitutions, we don't assume this. Instead we add the term formers for application (_ · _) and lambda-abstraction λ as natural transformations.

We start with a precise definition of a simply typed CwF with the additional structure to model simply typed λ-calculus (section 5.1) and then we show that the recursive definition of substitution gives rise to a simply typed CwF (section 5.2). We can define the initial CwF as a Quotient Inductive-Inductive Type (QIIT). To simplify our development, rather than using a Cubical Agda HIT, [7] we just postulate the existence of this QIIT in Agda (with the associated rewriting rules). By initiality, there is an evaluation functor from the initial CwF to the recursively defined CwF (defined in section 5.2). On the other hand, we can embed the recursive CwF into the initial CwF; this corresponds to the embedding of normal forms into λ-terms, only that here we talk about *substitution normal forms*. We then show that these two structure maps are inverse to each other and hence that the recursively defined CwF is indeed initial (section 5.3). The two identities correspond to completeness and stability in the language of normalisation functions.

5.1 Simply Typed CwFs

We define a record to capture simply typed CWFs:

```
record CwF-simple : Set₁ where
```

We start with the category of contexts, using the same names as introduced previously:

```
field
  Con : Set
  _⊨_ : Con → Con → Set
  id   : Γ ⊨ Γ
  _∘_  : Δ ⊨ Θ → Γ ⊨ Δ → Γ ⊨ Θ
  id∘  : id ∘ δ ≡ δ
  ∘id  : δ ∘ id ≡ δ
  ∘∘  : (ξ ∘ θ) ∘ δ ≡ ξ ∘ (θ ∘ δ)
```

We introduce the set of types and associate a presheaf with each type:

```
Ty      : Set
_⊢_  : Con → Ty → Set
_[_] : Γ ⊢ A → Δ ⊨ Γ → Δ ⊢ A
[id] : (t [ id ]) ≡ t
[∘]     : t [ θ ] [ δ ] ≡ t [ θ ∘ δ ]
```

The category of contexts has a terminal object (the empty context):

```
▪  : Con
ε  : Γ ⊨ ▪
•–η : δ ≡ ε
```

[7] Cubical Agda still lacks some essential automation, e.g. integrating no-confusion properties into pattern matching.

Context extension resembles categorical products but mixing contexts and types:

```
_▷_  : Con → Ty → Con
_,_  : Γ ⊨ Δ → Γ ⊢ A → Γ ⊨ (Δ ▷ A)
π₀   : Γ ⊨ (Δ ▷ A) → Γ ⊨ Δ
π₁   : Γ ⊨ (Δ ▷ A) → Γ ⊢ A
▷-β₀ : π₀ (δ , t) ≡ δ
▷-β₁ : π₁ (δ , t) ≡ t
▷-η  : (π₀ δ , π₁ δ) ≡ δ
π₀∘  : π₀ (θ ∘ δ) ≡ π₀ θ ∘ δ
π₁∘  : π₁ (θ ∘ δ) ≡ (π₁ θ) [ δ ]
```

We can define the morphism part of the context extension functor as before:

```
_↑_ : Γ ⊨ Δ → ∀ A → Γ ▷ A ⊨ Δ ▷ A
δ ↑ A = (δ ∘ (π₀ id)) , π₁ id
```

We need to add the specific components for simply typed λ-calculus; we add the type constructors, the term constructors and the corresponding naturality laws:

```
field
  o     : Ty
  _⇒_   : Ty → Ty → Ty
  _·_   : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
  λ_    : Γ ▷ A ⊢ B → Γ ⊢ A ⇒ B
  ·[]   : (t · u) [ δ ] ≡ (t [ δ ]) · (u [ δ ])
  λ[]   : (λ t) [ δ ] ≡ λ (t [ δ ↑ _ ])
```

5.2 The CwF of recursive substitutions

We are building towards a proof of initiality for our recursive substitution syntax, but shall start by showing that our recursive substitution syntax obeys the specified CwF laws, specifically that CwF-simple can be instantiated with _⊢[_]_/_⊨[_]_. This will be more-or-less enough to implement the "normalisation" direction of our initial CwF ≃ recursive sub syntax isomorphism.

Most of the work to prove these laws was already done in 4 but there are a couple tricky details with fitting into the exact structure the CwF-simple record requires.

```
module CwF = CwF-simple
```

```
is-cwf : CwF-simple
is-cwf .CwF.Con = Con
```

We need to decide which type family to interpret substitutions into. In our first attempt, we tried to pair renamings/substitutions with their sorts to stay polymorphic:

```
record _⊨_ (Δ : Con) (Γ : Con) : Set where
  field
    sort : Sort
    tms  : Δ ⊨[ sort ] Γ
```

```
is-cwf .CwF._⊨_ = _⊨_
is-cwf .CwF.id = record {sort = V; tms = id}
```

Unfortunately, this approach quickly breaks. The CwF laws force us to provide a unique morphism to the terminal context (i.e. a unique weakening from the empty context).

```
is-cwf .CwF.▪ = ▪
is-cwf .CwF.ε = record {sort = ?; tms = ε}
is-cwf .CwF.•-η {δ = record {sort = q; tms = ε}} = ?
```

Our _ ⊨ _ record is simply too flexible here. It allows two distinct implementations: **record** {sort = V; tms = ε} and **record** {sort = T; tms = ε}. We are stuck!

Therefore, we instead fix the sort to T.

```
is-cwf : CwF-simple
is-cwf .CwF.Con = Con
is-cwf .CwF._⊨_ = _⊨[ T ]_
is-cwf .CwF.▪ = ▪
is-cwf .CwF.ε = ε
is-cwf .CwF.•-η {δ = ε} = refl
is-cwf .CwF._∘_ = _∘_
is-cwf .CwF.∘∘ = sym ∘∘
```

The lack of flexibility over sorts when constructing substitutions does, however, make identity a little trickier. id doesn't fit CwF.id directly as it produces a renaming Γ ⊨[V] Γ. We need the equivalent substitution Γ ⊨[T] Γ.

We first extend tm⊑ to sequences of variables/terms:

```
tm*⊑ : q ⊑ s → Γ ⊨[ q ] Δ → Γ ⊨[ s ] Δ
tm*⊑ q⊑s ε = ε
tm*⊑ q⊑s (σ , x) = tm*⊑ q⊑s σ , tm⊑ q⊑s x
```

And prove various lemmas about how tm* ⊑ coercions can be lifted outside of our substitution operators:

```
⊑∘   : tm*⊑ v⊑t xs ∘ ys ≡ xs ∘ ys
∘⊑   : xs ∘ tm*⊑ v⊑t ys ≡ xs ∘ ys
v[⊑] : i [ tm*⊑ v⊑t ys ] ≡ tm⊑ v⊑t i [ ys ]
t[⊑] : t [ tm*⊑ v⊑t ys ] ≡ t [ ys ]
⊑⁺   : tm*⊑ ⊑t xs ⁺ A ≡ tm*⊑ v⊑t (xs ⁺ A)
⊑↑   : tm*⊑ v⊑t xs ↑ A ≡ tm*⊑ v⊑t (xs ↑ A)
```

Most of these are proofs come out easily by induction on terms and substitutions so we skip over them. Perhaps worth noting though is that ⊑⁺ requires one new law relating our two ways of weakening variables.

```
suc[id⁺] : i [ id ⁺ A ] ≡ suc i A
suc[id⁺] {i = i} {A = A} =
  i [ id ⁺ A ]
  ≡⟨ ⁺-nat[]v {i = i} ⟩
  suc (i [ id ]) A
```

```
    ≡⟨ cong (λ j → suc j A) [id] ⟩
    suc i A ∎
  ⊑⁺ {xs = ε} = refl
  ⊑⁺ {xs = xs , x} = cong₂ _,_ ⊑⁺ (cong (`_) suc[id⁺])
```

We can now build an identity substitution by applying this coercion to the identity renaming.

```
is-cwf .CwF.id = tm*⊑ v⊑t id
```

The left and right identity CwF laws now take the form `tm*⊑ v⊑t id ∘ δ ≡ δ` and `δ ∘ tm*⊑ v⊑t id ≡ δ`. This is where we can take full advantage of the `tm*⊑` machinery; these lemmas let us reuse our existing `id∘`/`∘id` proofs!

```
is-cwf .CwF.id∘ {δ = δ} =
  tm*⊑ v⊑t id ∘ δ
  ≡⟨ ⊑∘ ⟩
  id ∘ δ
  ≡⟨ id∘ ⟩
  δ ∎
is-cwf .CwF.∘id {δ = δ} =
  δ ∘ tm*⊑ v⊑t id
  ≡⟨ ∘⊑ ⟩
  δ ∘ id
  ≡⟨ ∘id ⟩
  δ ∎
```

Similarly to substitutions, we must fix the sort of our terms to `T` (in this case, so we can prove the identity law - note that applying the identity substitution to a variable `i` produces the distinct term `` ` i``).

```
is-cwf .CwF.Ty            = Ty
is-cwf .CwF._⊢_           = _⊢[ T ]_
is-cwf .CwF._[_]          = _[_]
is-cwf .CwF.[∘] {t = t} = sym ([∘] {x = t})
is-cwf .CwF.[id] {t = t} =
  t [ tm*⊑ v⊑t id ]
  ≡⟨ t[⊑] {t = t} ⟩
  t [ id ]
  ≡⟨ [id] ⟩
  t ∎
```

Context extension and the associated laws are easy. We define projections `π₀ (δ , t) = δ` and `π₁ (δ , t) = t` standalone as these will be useful in the next section also.

```
is-cwf .CwF._▷_ = _▷_
is-cwf .CwF._,_ = _,_
is-cwf .CwF.π₀ = π₀
is-cwf .CwF.π₁ = π₁
is-cwf .CwF.▷-β₀ = refl
is-cwf .CwF.▷-β₁ = refl
```

```
is-cwf .CwF.▷-η {δ = xs , x} = refl
is-cwf .CwF.π₀∘ {θ = xs , x} = refl
is-cwf .CwF.π₁∘ {θ = xs , x} = refl
```

Finally, we can deal with the cases specific to simply typed λ-calculus. Only the β-rule for substitutions applied to lambdas is non-trivial due to differing implementations of _↑_.

```
is-cwf .CwF.o = o
is-cwf .CwF._⇒_ = _⇒_
is-cwf .CwF._·_ = _·_
is-cwf .CwF.λ_ = λ_
is-cwf .CwF.·[] = refl
is-cwf .CwF.λ[] {A = A} {t = x} {δ = ys} =
  λ x [ ys ↑ A ]
  ≡⟨ cong (λ ρ → λ x [ ρ ↑ A ]) (sym ∘id) ⟩
  λ x [ (ys ∘ id) ↑ A ]
  ≡⟨ cong (λ ρ → λ x [ ρ , ` zero ]) (sym ^+–nat∘) ⟩
  λ x [ ys ∘ id ^+ A , ` zero ]
  ≡⟨ cong (λ ρ → λ x [ ρ , ` zero ])
    (sym (∘⊑ {ys = id ^+ _})) ⟩
  λ x [ ys ∘ tm*⊑ v⊑t (id ^+ A) , ` zero ] ■
```

We have shown our recursive substitution syntax satisfies the CwF laws, but we want to go a step further and show initiality: that our syntax is isomorphic to the initial CwF.

An important first step is to actually define the initial CwF (and its eliminator). We use postulates and rewrite rules instead of a Cubical Agda higher inductive type (HIT) because of technical limitations mentioned previously. We also reuse our existing datatypes for contexts and types for convenience (note terms do not occur inside types in STLC).

To state the dependent equations between outputs of the eliminator, we need dependent identity types. We can define this simply by matching on the identity between the LHS and RHS types.

```
_≡[_]≡_ : ∀ {A B : Set ℓ} → A → A ≡ B → B
  → Set ℓ
x ≡[ refl ]≡ y = x ≡ y
```

To avoid name clashes between our existing syntax and the initial CwF constructors, we annotate every ICwF constructor with $^{\mathrm{I}}$.

```
postulate
  _⊢^I_ : Con → Ty → Set
  _⊨^I_ : Con → Con → Set
  id^I : Γ ⊨^I Γ
  _∘^I_ : Δ ⊨^I Γ → Θ ⊨^I Δ → Θ ⊨^I Γ
  id∘^I : id^I ∘^I δ^I ≡ δ^I
    -- ...
```

We state the eliminator for the initial CwF in terms of Motive and Methods records as in [4].

```
record Motive : Set₁ where
  field
```

```
    Con^M  : Con → Set
    Ty^M   : Ty → Set
    Tm^M   : Con^M Γ → Ty^M A → Γ ⊢^I A → Set
    Tms^M  : Con^M Δ → Con^M Γ → Δ ⊨^I Γ → Set

record Methods (𝕄 : Motive) : Set_1 where
  field
    id^M : Tms^M Γ^M Γ^M id^I
    _∘^M_ : Tms^M Δ^M Γ^M σ^I → Tms^M θ^M Δ^M δ^I
      → Tms^M θ^M Γ^M (σ^I ∘^I δ^I)
    id∘^M : id^M ∘^M δ^M ≡[ cong (Tms^M Δ^M Γ^M) id∘^I ]≡ δ^M
      -- ...

module Eliminator {𝕄} (𝐦 : Methods 𝕄) where
  open Motive 𝕄
  open Methods 𝐦
  elim-con : ∀ Γ → Con^M Γ
  elim-ty : ∀ A → Ty^M A
  elim-con • = •^M
  elim-con (Γ ▷ A) = (elim-con Γ) ▷^M (elim-ty A)
  elim-ty o = o^M
  elim-ty (A ⇒ B) = (elim-ty A) ⇒^M (elim-ty B)
  postulate
    elim-cwf : ∀ t^I → Tm^M (elim-con Γ) (elim-ty A) t^I
    elim-cwf* : ∀ δ^I → Tms^M (elim-con Δ) (elim-con Γ) δ^I
    elim-cwf*-idβ : elim-cwf* (id^I {Γ}) ≡ id^M
    elim-cwf*-∘β : elim-cwf* (σ^I ∘^I δ^I)
                 ≡ elim-cwf* σ^I ∘^M elim-cwf* δ^I
      -- ...

   {-# REWRITE elim-cwf*-idβ #-}
   {-# REWRITE elim-cwf*-∘β #-}
  -- ...
```

Normalisation from the initial CwF into substitution normal forms now only needs a way to connect our notion of "being a CwF" with our initial CwF's eliminator: specifically, that any set of type families satisfying the CwF laws gives rise to a Motive and associated set of Methods.

The one extra ingredient we need to make this work out neatly is to introduce a new reduction for cong: [8]

```
cong-const : ∀ {x : A} {y z : B} {p : y ≡ z}
  → cong (λ _ → x) p ≡ refl
cong-const {p = refl} = refl
```

[8] This definitional identity also holds natively in Cubical.

```
{-# REWRITE cong-const #-}
```

This enables the no-longer-dependent _ ≡[_]≡ _s to collapse to _ ≡ _s automatically.

```
module Recursor (cwf : CwF-simple) where
  cwf-to-motive : Motive
  cwf-to-methods : Methods cwf-to-motive

  rec-con  = elim-con cwf-to-methods
  rec-ty   = elim-ty  cwf-to-methods
  rec-cwf  = elim-cwf cwf-to-methods
  rec-cwf∗ = elim-cwf∗ cwf-to-methods

  cwf-to-motive .Conᴹ _      = cwf .CwF.Con
  cwf-to-motive .Tyᴹ _       = cwf .CwF.Ty
  cwf-to-motive .Tmᴹ Γ A _   = cwf .CwF._⊢_ Γ A
  cwf-to-motive .Tmsᴹ Δ Γ _  = cwf .CwF._⊨_ Δ Γ

  cwf-to-methods .idᴹ     = cwf .CwF.id
  cwf-to-methods ._∘ᴹ_    = cwf .CwF._∘_
  cwf-to-methods .id∘ᴹ    = cwf .CwF.id∘
    -- ...
```

Normalisation into our substitution normal forms can now be achieved by with:

```
norm : Γ ⊢ᴵ A → rec-con is-cwf Γ ⊢[ T ] rec-ty is-cwf A
norm = rec-cwf is-cwf
```

Of course, normalisation shouldn't change the type of a term, or the context it is in, so we might hope for a simpler signature Γ ⊢ᴵ A → Γ ⊢[T] A and, conveniently, rewrite rules can get us there!

```
Con≡ : rec-con is-cwf Γ ≡ Γ
Ty≡  : rec-ty is-cwf A ≡ A

Con≡ {Γ = •} = refl
Con≡ {Γ = Γ ▷ A} = cong₂ _▷_ Con≡ Ty≡

Ty≡ {A = o} = refl
Ty≡ {A = A ⇒ B} = cong₂ _⇒_ Ty≡ Ty≡
```

```
{-# REWRITE Con≡ Ty≡ #-}
```

```
norm : Γ ⊢ᴵ A → Γ ⊢[ T ] A
norm = rec-cwf is-cwf

norm∗ : Δ ⊨ᴵ Γ → Δ ⊨[ T ] Γ
norm∗ = rec-cwf∗ is-cwf
```

The inverse operation to inject our syntax back into the initial CwF is easily implemented by recursing on our substitution normal forms.

```
⌜_⌝ : Γ ⊢[ q ] A → Γ ⊢ᴵ A
⌜ zero ⌝    = zeroᴵ
⌜ suc i B ⌝ = sucᴵ ⌜ i ⌝ B
```

```
⌜ ` i ⌝     = ⌜ i ⌝
⌜ t · u ⌝ = ⌜ t ⌝ ·ᴵ ⌜ u ⌝
⌜ ƛ t ⌝     = ƛᴵ ⌜ t ⌝

⌜_⌝* : Δ ⊨[ q ] Γ → Δ ⊨ᴵ Γ
⌜ ε ⌝* = εᴵ
⌜ δ , x ⌝* = ⌜ δ ⌝* ,ᴵ ⌜ x ⌝
```

5.3 Proving initiality

We have implemented both directions of the isomorphism. Now to show this truly is an isomorphism and not just a pair of functions between two types, we must prove that norm and ⌜_⌝ are mutual inverses - i.e. stability (norm ⌜ t ⌝ ≡ t) and completeness (⌜ norm t ⌝ ≡ t).

We start with stability, as it is considerably easier. There are just a couple details worth mentioning:

- To deal with variables in the `_ case, we phrase the lemma in a slightly more general way, taking expressions of any sort and coercing them up to sort T on the RHS.
- The case for variables relies on a bit of coercion manipulation and our earlier lemma equating i [id ⁺ B] and suc i B.

```
stab : norm ⌜ x ⌝ ≡ tm⊑ ⊑t x
stab {x = zero} = refl
stab {x = suc i B} =
  norm ⌜ i ⌝ [ tm*⊑ v⊑t (id ⁺ B) ]
  ≡⟨ t[⊑] {t = norm ⌜ i ⌝} ⟩
  norm ⌜ i ⌝ [ id ⁺ B ]
  ≡⟨ cong (λ j → suc[ _ ] j B) (stab {x = i}) ⟩
  ` i [ id ⁺ B ]
  ≡⟨ cong `_ suc[id⁺] ⟩
  ` suc i B ∎
stab {x = ` i} = stab {x = i}
stab {x = t · u} =
  cong₂ _·_ (stab {x = t}) (stab {x = u})
stab {x = ƛ t} = cong ƛ_ (stab {x = t})
```

To prove completeness, we must instead induct on the initial CwF itself, which means there are many more cases. We start with the motive:

```
compl-𝕄 : Motive
compl-𝕄 .Conᴹ _ = ⊤
compl-𝕄 .Tyᴹ _ = ⊤
compl-𝕄 .Tmᴹ _ _ tᴵ = ⌜ norm tᴵ ⌝ ≡ tᴵ
compl-𝕄 .Tmsᴹ _ _ δᴵ = ⌜ norm* δᴵ ⌝* ≡ δᴵ
```

To show these identities, we need to prove that our various recursively defined syntax operations are preserved by ⌜_⌝.

Preservation of zero[_] reduces to reflexivity after splitting on the sort.

```
⌜zero⌝ : ⌜ zero[_] {Γ = Γ} {A = A} q ⌝ ≡ zeroᴵ
⌜zero⌝ {q = V} = refl
⌜zero⌝ {q = T} = refl
```

Preservation of each of the projections out of sequences of terms (e.g. $\ulcorner \pi_0\ \delta \urcorner * \ \equiv\ \pi_0^{\mathrm{I}} \ulcorner \delta \urcorner *$) reduce to the associated β-laws of the initial CwF (e.g. $\triangleright{-}\beta_0^{\mathrm{I}}$).

Preservation proofs for _[_], _ ↑ _, _⁺_, id and suc[_] are all mutually inductive, mirroring their original recursive definitions. We must stay polymorphic over sorts and again use our dummy Sort argument trick when implementing ⌜id⌝ to keep Agda's termination checker happy.

```
⌜[]⌝  : ⌜ x [ ys ] ⌝ ≡ ⌜ x ⌝ [ ⌜ ys ⌝* ]ᴵ
⌜↑⌝  : ⌜ xs ↑ A ⌝* ≡ ⌜ xs ⌝* ↑ᴵ A
⌜+⌝  : ⌜ xs ⁺ A ⌝* ≡ ⌜ xs ⌝* ∘ᴵ wkᴵ
⌜id⌝ : ⌜ id {Γ = Γ} ⌝* ≡ idᴵ
⌜suc⌝ : ⌜ suc[ q ] x B ⌝ ≡ ⌜ x ⌝ [ wkᴵ ]ᴵ
⌜id⌝′ : Sort → ⌜ id {Γ = Γ} ⌝* ≡ idᴵ
⌜id⌝ = ⌜id⌝′ V
 {-# INLINE ⌜id⌝ #-}
```

To complete these proofs, we also need β-laws about our initial CwF substitutions, so we derive these now.

```
zero[]ᴵ : zeroᴵ [ δᴵ ,ᴵ tᴵ ]ᴵ ≡ tᴵ
zero[]ᴵ {δᴵ = δᴵ} {tᴵ = tᴵ} =
  zeroᴵ [ δᴵ ,ᴵ tᴵ ]ᴵ
  ≡⟨ sym π₁∘ᴵ ⟩
  π₁ᴵ (idᴵ ∘ᴵ (δᴵ ,ᴵ tᴵ))
  ≡⟨ cong π₁ᴵ id∘ᴵ ⟩
  π₁ᴵ (δᴵ ,ᴵ tᴵ)
  ≡⟨ ▷-β₁ᴵ ⟩
  tᴵ ∎
```

```
suc[]ᴵ : sucᴵ tᴵ B [ δᴵ ,ᴵ uᴵ ]ᴵ ≡ tᴵ [ δᴵ ]ᴵ
suc[]ᴵ =  -- ...
,[]ᴵ : (δᴵ ,ᴵ tᴵ) ∘ᴵ σᴵ ≡ (δᴵ ∘ᴵ σᴵ) ,ᴵ (tᴵ [ σᴵ ]ᴵ)
,[]ᴵ =  -- ...
```

We also need a couple lemmas about how ⌜_⌝ treats terms of different sorts identically.

```
⌜⊑⌝ : ∀ {x : Γ ⊢[ q ] A} → ⌜ tm⊑ ⊑t x ⌝ ≡ ⌜ x ⌝
⌜⊑⌝* : ⌜ tm*⊑ ⊑t xs ⌝* ≡ ⌜ xs ⌝*
```

We can now (finally) proceed with the proofs. There are quite a few cases to cover, so for brevity we elide the proofs of ⌜[]⌝ and ⌜suc⌝.

```
⌜↑⌝ {q = q} = cong₂ _,ᴵ_ ⌜+⌝ (⌜zero⌝ {q = q})
⌜+⌝ {xs = ε} = sym •-ηᴵ
⌜+⌝ {xs = xs , x} {A = A} =
  ⌜ xs ⁺ A ⌝* ,ᴵ ⌜ suc[ _ ] x A ⌝
  ≡⟨ cong₂ _,ᴵ_ ⌜+⌝ (⌜suc⌝ {x = x}) ⟩
  (⌜ xs ⌝* ∘ᴵ wkᴵ) ,ᴵ (⌜ x ⌝ [ wkᴵ ]ᴵ)
```

```
    ≡⟨ sym ,[]^I ⟩
    (⌜ xs ⌝* ,^I ⌜ x ⌝) ∘^I wk^I ■
  ⌜id⌝′ {Γ = •} _ = sym •-η^I
  ⌜id⌝′ {Γ = Γ ▷ A} _ =
    ⌜ id ⁺ A ⌝* ,^I zero^I
    ≡⟨ cong (_,^I zero^I) ⌜+⌝ ⟩
    ⌜ id ⌝* ↑^I A
    ≡⟨ cong (_^^I A) ⌜id⌝ ⟩
    id^I ↑^I A
    ≡⟨ cong (_,^I zero^I) id∘^I ⟩
    wk^I ,^I zero^I
    ≡⟨ ▷-η^I ⟩
    id^I ■
```

We also prove preservation of substitution composition $\ulcorner\circ\urcorner : \ulcorner \mathsf{xs} \circ \mathsf{ys} \urcorner{*} \equiv \ulcorner \mathsf{xs} \urcorner{*} \circ^{I} \ulcorner \mathsf{ys} \urcorner{*}$ in similar fashion.

The main cases of Methods compl-$\mathbb{M}$ can now be proved by just applying the preservation lemmas and inductive hypotheses.

```
  compl-m : Methods compl-𝕄
  compl-m .id^M =
    ⌜ tm*⊑ v⊑t id ⌝*
    ≡⟨ ⌜⊑⌝* ⟩
    ⌜ id ⌝*
    ≡⟨ ⌜id⌝ ⟩
    id^I ■
  compl-m ._∘^M_ {σ^I = σ^I} {δ^I = δ^I} σ^M δ^M =
    ⌜ norm* σ^I ∘ norm* δ^I ⌝*
    ≡⟨ ⌜∘⌝ ⟩
    ⌜ norm* σ^I ⌝* ∘^I ⌜ norm* δ^I ⌝*
    ≡⟨ cong₂ _∘^I_ σ^M δ^M ⟩
    σ^I ∘^I δ^I ■
    -- ...
```

The remaining cases correspond to the CwF laws, which must hold for whatever type family we eliminate into in order to retain congruence of $\_ \equiv \_$. In our completeness proof, we are eliminating into equations, and so all of these cases are higher identities (demanding we equate different proof trees for completeness, instantiated with the LHS/RHS terms/substitutions).

In a univalent type theory, we might try and carefully introduce additional coherences to our initial CwF to try and make these identities provable without the sledgehammer of set truncation (which prevents eliminating the initial CwF into any non-set).

As we are working in vanilla Agda, we'll take a simpler approach, and rely on UIP (duip : ∀ {x y z w r} {p : x ≡ y} {q : z ≡ w} → p ≡[r]≡ q).[9]

---

[9] Note that proving this form of (dependent) UIP relies on type constructor injectivity (specifically, injectivity of $\_ \equiv \_$). We could use a weaker version taking an additional proof of x ≡ z, but this would be clunkier to use; Agda has no hope of inferring such a proof by unification.

```
compl-m .id ∘ᴹ  = duip
compl-m . ∘idᴹ  = duip
  -- ...
```

And completeness is just one call to the eliminator away.

```
compl : ⌜ norm tᴵ ⌝ ≡ tᴵ
compl {tᴵ = tᴵ} = elim-cwf compl-m tᴵ
```

6 Conclusions and further work

The subject of the paper is a problem which everybody (including ourselves) would have thought to be trivial. As it turns out, it isn't, and we spent quite some time going down alleys that didn't work. With hindsight, the main idea seems rather obvious: introduce sorts as a datatype with the structure of a boolean algebra. To implement the solution in Agda, we managed to convince the termination checker that V is structurally smaller than T and so left the actual work determining and verifying the termination ordering to Agda. This greatly simplifies the formal development.

We could, however, simplify our development slightly further if we were able to instrument the termination checker, for example with an ordering on constructors (i.e. removing the need for the T>V encoding). We also ran into issues with Agda only examining direct arguments to function calls for identifying termination order. The solutions to these problems were all quite mechanical, which perhaps implies there is room for Agda's termination checking to be extended. Finally, it would be nice if the termination checker provided independently-checkable evidence that its non-trivial reasoning is sound (being able to print termination matrices with -v term:5 is a useful feature, but is not quite as convincing as actually elaborating to well-founded induction like e.g. Lean).

It is perhaps worth mentioning that the convenience of our solution heavily relies on Agda's built-in support for lexicographic termination [2]. This is in contrast to Rocq and Lean; the former's Fixpoint command merely supports structural recursion on a single argument and the latter has only raw elimination principles as primitive. Luckily, both of these proof assistants layer on additional commands/tactics to support more natural use of non-primitive induction.

For example, Lean features a pair of tactics termination_by and decreasing_by for specifying per-function termination measures and proving that these measures strictly decrease, similarly to our approach to justifying termination in 3.1. The slight extra complication is that Lean requires the provided measures to strictly decrease along every mutual function call as opposed to over every cycle in the call graph. In the case of our substitution operations, adapting for this is not to onerous, requiring e.g. replacing the measures for id and $\_^{+}\_$ from $(r_2\ ,\ \Gamma_2)$ and $(r_3\ ,\ \sigma_3)$ to $(r_2\ ,\ \Gamma_2\ ,\ 0)$ and $(r_3\ ,\ 0\ ,\ \sigma_3)$, ensuring a strict decrease when calling $\_^{+}\_$ in id $\{\Gamma\ =\ \Gamma\ \rhd\ A\}$.

Conveniently, after specifying the correct measures, Lean is able to automatically solve the decreasing_by proof obligations, and so our approach to defining substitution remains concise even without quite-as-robust support for lexicographic termination[10]. Of course,

[10] In fact, specifying termination measures manually has some advantages: we no longer need to use a complicated Sort datatype to make the ordering on constructors obvious: computing sizes with if b then 1 else 0 is sufficient.

doing the analysis to work out which termination measures were appropriate took some time, and one could imagine an expanded Lean tactic being able to infer termination with no assistance, using a similar algorithm to Agda.

We could avoid a recursive definition of substitution altogether and only work with the initial simply typed CwF as a QIIT. However, this is unsatisfactory for two reasons: first of all, we would like to relate the quotiented view of λ-terms to the their definitional presentation, and, second, when proving properties of λ-terms it is preferable to do so by induction over terms rather than use quotients (i.e. no need to consider cases for non-canonical elements or prove that equations are preserved).

One reviewer asked about another alternative: since we are merging _ ∋ _ and _ ⊢ _ why not go further and merge them entirely? Instead of a separate type for variables, one could have a term corresponding to de Bruijn index zero (written • below) and an explicit weakening operator on terms (written _↑).

```
data _⊢′_ : Con → Ty → Set where
  •     : Γ ▷ A ⊢′ A
  _↑    : Γ ⊢′ B → Γ ▷ A ⊢′ B
  _·_   : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
  λ_    : Γ ▷ A ⊢ B → Γ ⊢ A ⇒ B
```

This has the unfortunate property that there is now more than one way to write terms that used to be identical. For instance, the terms • ↑ ↑ · • ↑ · • and (• ↑ · •) ↑ · • are equivalent, where • ↑ ↑ corresponds to the variable with de Bruijn index two. A development along these lines is explored in [19]. It leads to a compact development, but one where the natural normal form appears to be to push weakening to the outside (such as in [14]), so that the second of the two terms above is considered normal rather than the first. It may be a useful alternative, but we think it is also interesting to pursue the development given here, where terms retain their familiar normal form.

This paper can also be seen as a preparation for the harder problem to implement recursive substitution for dependent types. This is harder, because here the typing of the constructors actually depends on the substitution laws. While such a Münchhausian [5] construction [11] should actually be possible in Agda, the theoretical underpinning of inductive-inductive-recursive definitions is mostly unexplored (with the exception of the proposal by [11]). However, there are potential interesting applications: strictifying substitution laws is essential to prove coherence of models of type theory in higher types, in the sense of HoTT.

Hence this paper has two aspects: it turns out that an apparently trivial problem isn't so easy after all, and it is a stepping stone to more exciting open questions. But before you can run you need to walk and we believe that the construction here can be useful to others.

References

1. Andreas Abel. Parallel substitution as an operation for untyped de bruijn terms. Agda proof, 2011.
2. Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.
3. Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 195–207, 2017.

[11] The reference is to Baron Münchhausen, who allegedly pulled himself out of a swamp by his own hair.

4 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *SIGPLAN Not.*, 51(1):18–29, jan 2016. `doi:10.1145/2914770.2837638`.

5 Thorsten Altenkirch, Ambrus Kaposi, Artjoms Šinkarovs, and Tamás Végh. The münchhausen method in type theory. In *28th International Conference on Types for Proofs and Programs 2022*, page 10. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

6 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453–468, 1999.

7 Thosten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical methods in computer science*, 11, 2015.

8 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, pages 135–180, 2021.

9 Haskell Brooks Curry and Robert Feys. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

10 N. G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972. URL: `https://www.sciencedirect.com/science/article/pii/1385725872900340`, `doi:10.1016/1385-7258(72)90034-0`.

11 Ambrus Kaposi. Towards quotient inductive-inductive-recursive types. In *29th International Conference on Types for Proofs and Programs TYPES 2023–Abstracts*, page 124, 2023.

12 Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, pages 3–10, 2010.

13 Conor McBride. Type-preserving renaming and substitution. *Journal of Functional Programming*, 2006.

14 Conor McBride. Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science*, 275:53–69, July 2018. Mathematically Structured Functional Programming, MSFP ; Conference date: 08-07-2018 Through 08-07-2018. URL: `https://msfp2018.bentnib.org/`, `doi:10.4204/EPTCS.275.6`.

15 Hannes Saffrich. Abstractions for multi-sorted substitutions. In *15th International Conference on Interactive Theorem Proving (ITP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

16 Hannes Saffrich, Peter Thiemann, and Marius Weidner. Intrinsically typed syntax, a logical relation, and the scourge of the transfer lemma. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development*, pages 2–15, 2024.

17 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 166–180, 2019.

18 The Agda Team. Agda documentation. `https://agda.readthedocs.io`, 2024. Accessed: 2024-08-26.

19 Philip Wadler. Explicit weakening. *Electronic Proceedings in Theoretical Computer Science*, 413:15–26, November 2024. Festschrift for Peter Thiemann. URL: `http://arxiv.org/abs/2412.03124`, `doi:10.4204/EPTCS.413.2`.