Substitution Without Copy and Paste

Thorsten Altenkirch
University of Nottingham
Nottingham, UK
thorsten.altenkirch@nottingham.ac.uk

Nathaniel Burke
Imperial College London
London, UK
nathanielrburke3@gmail.com

Philip Wadler
University of Edinburgh and Input Output
Edinburgh, UK
wadler@inf.ed.ac.uk

Defining substitution for a language with binders like the simply typed λ-calculus requires repetition, defining substitution and renaming separately. To verify the categorical properties of this calculus, we must repeat the same argument many times. We present a lightweight method that avoids repetition and that gives rise to a simply typed category with families (CwF) isomorphic to the initial simply typed CwF. Our paper is a literate Agda script.

1 Introduction

> Some half dozen persons have written technically on combinatory logic, and most of these, including ourselves, have published something erroneous. [14]

The first author was writing an introduction to category theory for functional programmers. One example was the category of simply-typed λ-terms and substitutions, and proving the expected category laws seemed a suitable exercise. We attempted to mechanise the solution in Agda [23], and hit a setback: multiple proofs had to be repeated multiple times. A guideline of good software engineering is to **not write code by copy and paste**, and this applies doubly to formal proofs.

This paper is the result of our effort to refactor the proof. The method used also applies to other problems; in particular, we see the current construction as a warmup for the definition of substitution for dependent type theory, which may have interesting applications for interpreting dependent types in higher categories (coherence).

1.1 In a nutshell

When working with substitution for a calculus with binders, we have to differentiate between renamings ($\Delta \Vdash^{\mathsf{v}} \Gamma$), where variables are substituted only for variables ($\Gamma \ni A$), and proper substitutions ($\Delta \Vdash \Gamma$), where variables are replaced with terms ($\Gamma \vdash A$). This results in several similar operations:

$$\_^{\mathsf{v}}[\_]^{\mathsf{v}} : \Gamma \ni A \to \Delta \Vdash^{\mathsf{v}} \Gamma \to \Delta \ni A \qquad \_[\_]^{\mathsf{v}} : \Gamma \vdash A \to \Delta \Vdash^{\mathsf{v}} \Gamma \to \Delta \vdash A$$
$$\_^{\mathsf{v}}[\_] : \Gamma \ni A \to \Delta \Vdash \Gamma \to \Delta \vdash A \qquad \_[\_] : \Gamma \vdash A \to \Delta \Vdash \Gamma \to \Delta \vdash A$$

The duplication gets worse when we prove properties of substitution, such as the functor law

$$x\,[\,xs \circ ys\,] \equiv x\,[\,xs\,]\,[\,ys\,]$$

Chaudhuri, Nantes-Sobrinho (Eds.): International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2025)
EPTCS ??, 2025, pp. 1–17, doi:10.4204/EPTCS.??.??

All components x, xs, ys can be either variables/renamings or terms/substitutions, so we must prove eight combinations. The repetition extends to the intermediary lemmas.

Our solution is to introduce a type of sorts with $\mathsf{V} : \mathsf{Sort}$ for variables/renamings and $\mathsf{T} : \mathsf{Sort}$ for terms/substitutions, leading to a single substitution operation

$$\_[\_] : \Gamma \vdash [\, \mathsf{q} \,]\, \mathsf{A} \to \Delta \Vdash [\, \mathsf{r} \,]\, \Gamma \to \Delta \vdash [\, \mathsf{q} \sqcup \mathsf{r} \,]\, \mathsf{A}$$

where $\mathsf{q}, \mathsf{r} : \mathsf{Sort}$ and $\mathsf{q} \sqcup \mathsf{r}$ is the least upper bound in the lattice of sorts with $\mathsf{V} \sqsubseteq \mathsf{T}$. Now we need only prove one variant of the functor law, relying on the fact that $\_ \sqcup \_$ is associative. Our mutually recursive definitions are accepted by Agda, as we can convince its termination checker that V is structurally smaller than T (see Section 3).

As a specification, we formulate an explicit substitution calculus as a quotient-inductive type, or QIT (a mutual inductive type with equations). Here, substitution itself becomes a term former. In our specification, the substitution laws correspond to the equations of a simply-typed category with families (CwF)—a variant of a CwF where the types do not depend on a context. Our recursive substitution operations lead to a simply typed CwF isomorphic to the initial one, yielding a normalisation result where λ-terms without explicit substitutions are *substitution normal forms*.

1.2 Related work

De Bruijn introduced his eponymous indices and simultaneous substitution in [12]. We use typed typed de Bruijn indices as in [9].

In [9], termination of substitution was shown using well-founded recursion. Our approach is simpler and scales better. Andreas Abel used a similar technique to ours to mechanise [9], without manual well-founded recursion, in an unpublished Agda proof [1].

The duplication between renaming and substitution operations is factored into *kits* in [18]. In [5], it was further shown how to extend this factoring to the proofs (by developing a "fusion framework"). In languages supporting lexicographic recursion, our technique is simpler.

All the works listed so far also embrace the monadic perspective. That is, encoding substitutions as functions from variables to terms (indeed, this is one of the motivations for relative monads [6]). However, it is not clear how to extend this approach to dependently typed languages without "very dependent" [15, 8] function types.

There have been a number of other publications on mechanising substitution. Schäfer and Stark *et al* [21, 22] develop a Rocq library which automatically derives substitution lemmas, but the proofs are repeated for renamings and substitutions (as in Section 2). Their equational theory is also similar to the simply typed CwFs in Section 5. Saffrich [19] uses Agda with an *extrinsic* formulation (with preterms and typing separate), and applies [5] to factor the construction using kits. In contrast, Saffrich [20] uses Agda with an *intrinsic* formulation (as here, indexing terms by types), but defines renaming and substitution separately, and the relevant substitution lemmas are repeated for all required combinations.

2 The naive approach

First, we review the copy-and-paste approach. 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
```

The constructor `_ embeds variables in λ-terms and we write applications as t · u. Following de Bruijn, lambda abstraction λ_ doesn't bind a name explicitly. Instead, variables count the number of binders between them and their binding site. Substitutions (ts, us, vs) are sequences of terms:

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

Now we attempt to define the action of 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 ?
```

We encounter a problem with the case for binders λ_. We are given a substitution ts : Δ ⊩ Γ but the body lives in the extended context t : Γ , A ⊢ B. We exploit functoriality of context extension (_ ▷ _), _↑_ : Γ ⊩ Δ → (A : Ty) → Γ ▷ A ⊩ Δ ▷ A, and write (λ t) [ts] = λ (t [ts ↑ _]).

Now, we must define _↑_. This is easy (isn't it?), but we need weakening of substitutions (_⁺_):

```
ts ↑ A = ts ⁺ A , ` zero
```

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

Which, in turn, is just a fold of term-weakening (suc-tm) over substitutions:

```
ε       ⁺ A = ε
(ts , t) ⁺ A = ts ⁺ A , suc-tm t A
```

```
suc-tm : Γ ⊢ B → (A : Ty) → Γ ▷ A ⊢ B
```

But how can we define suc-tm when we only have weakening for variables (vs)? If we already had identity id : Γ ⊩ Γ and substitution we could write: suc-tm t A = t [id ⁺ A], but this is not structurally recursive (and is rejected by Agda's termination checker).

To fix this, we use that id is a renaming, i.e. a substitution only containing variables, and defining _⁺v_ for renamings is easy. This leads to a structurally recursive definition, though with some repetition.

```
data _⊩v_ : Con → Con → Set where
  ε : Γ ⊩v •
  _,_ : Γ ⊩v Δ → Γ ∋ A → Γ ⊩v Δ ▷ A
```

```
_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 ⊩v Δ → ∀ A → Γ ▷ A ⊩v Δ
ε       ⁺v A = ε
(is , i) ⁺v A = is ⁺v A , suc i A
```

```
_↑v_ : Γ ⊩v Δ → ∀ A → Γ ▷ A ⊩v Δ ▷ A
is ↑v A = is ⁺v A , zero
```

```
_[_]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 seem too bad, but it gets worse when proving the laws. Let _∘_ be composition of substitutions. To prove associativity, we first need functoriality, [∘] : t [us ∘ vs] ≡ t [us] [vs] but to prove this, we also need to cover all variations where t, us, vs are variables/renamings rather than terms/substitutions.

This leads to eight combinations, with the cases for each constructor of t reading near-identically. This repetition is emblematic of many prior attempts at mechanising substitution [4, 11, 21, 22, 20].

The rest of the paper shows how to factor these definitions and proofs, using only (lexicographic) structural recursion.

3 Factorising with sorts

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

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

But this is not quite what we want. Agda's termination checker uses structural orderings. Following our intuition that variable weakening is trivial but term weakening requires renaming, we would like the sort of variables V to be structurally smaller than the sort of terms T.

With the following definition, we make V structurally smaller than T>V V isV, while maintaining that Sort has only two elements.

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

data IsV : Sort → Set where
  isV : IsV V
```

The predicate isV only holds for V. This encoding makes use of Agda's support for inductive-inductive datatypes (IITs), but a pair of a natural number n and a proof $n \leqslant 1$ would also work, i.e. $\mathsf{Sort} = \Sigma\,\mathbb{N}\,(\_ \leqslant 1)$. We make T : Sort an abbreviation for T>V V isV with a pattern declaration.

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

Now we can pattern match over Sort with V and T, while Agda's termination checker treats V as structurally smaller than T.

It is now possible to define terms and variables (x, y, z) in one go:

```
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
  ƛ_   : Γ ▷ A ⊢[ T ] B → Γ ⊢[ T ] A ⇒ B
```

This recapitulates our previous definitions (in Section 2), where $\Gamma \vdash [\ \mathsf{V}\ ]\ \mathsf{A}$ corresponds to $\Gamma \ni \mathsf{A}$ and $\Gamma \vdash [\ \mathsf{T}\ ]\ \mathsf{A}$ to $\Gamma \vdash \mathsf{A}$. Now we can parametrise our previous development. As a first step, we generalise 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
```

We model the structural order on sorts as an explicit relation with a least upper bound. The latter will help with substitution and composition, where the result is V only if both inputs are V.

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

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

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
⊔t : q ⊔ T ≡ T
```

These are easy to prove by case analysis, e.g. $\sqsubseteq t\,\{V\} = v\sqsubseteq t$ and $\sqsubseteq t\,\{T\} = \mathsf{rfl}$.

Further, we turn the equations (⊔⊔, ⊔v, ⊔t) into rewrite rules with {-# **REWRITE** ⊔⊔ ⊔v ⊔t #-}. This introduces new definitional equalities, allowing the type checker to directly exploit e.g. associativity of _⊔_ (effectively, this feature allows a selective use of extensional type theory).

Functoriality of context extension is now parametric

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

We'll derive this later. Meanwhile, the order on sorts gives rise to another functorial action.

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

Now we can define substitution and renaming in one go:

```
_[_] : 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 ↑ _ ])
```

Here _⊔_ ensures substitution returns a variable only if both inputs are variables/renamings. We use tm⊑ when substituting for variables because i [xs] will return a variable if xs is a renaming, but (` i) [xs] must return a term.

We define id using _↑_, recursing over contexts. To define _↑_ itself, we need parametric versions of zero and suc. Defining zero is easy.

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

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

However, suc is more subtle since the case for T depends on weakening over substitutions:

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

```
_⁺_ :  Γ ⊩[ q ] Δ → ∀ A
    → Γ ▷ A ⊩[ q ] Δ
ε       ⁺ A = ε
(xs , x) ⁺ A = xs ⁺ A , suc[ _ ] x A
```

And finally we can define _↑_ and _∘_.

```
xs ↑ A = xs ⁺ A , zero[ _ ]
```

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

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) applying a renaming rather than a full substitution. Note that if instead we had id : Γ ⊩[T] Γ, or if weakening for variables (suc[V] i A) was implemented by i [id ⁺ A], our operations would still be type-correct but would genuinely loop, so perhaps Agda is right to be careful.

We have appropriately specialised weakening for variables though, so why doesn't Agda accept our program? The limitation is ultimately technical: Agda only looks at direct arguments to function calls when building the call graph from which it identifies termination order [3]. Because id is not passed a sort, the sort cannot be considered as decreasing in the case of term weakening (suc[T] t A).

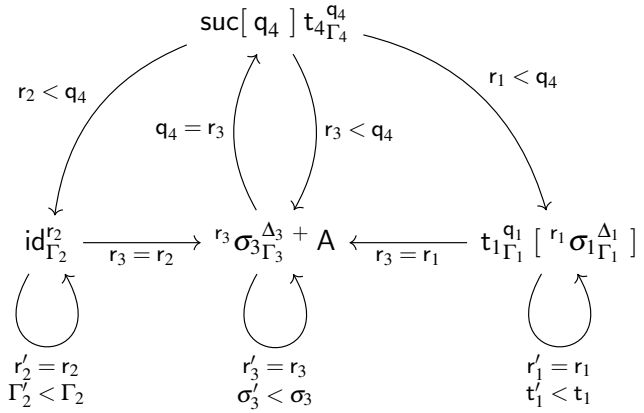

Figure 1: Call graph of substitution operations

Function	Measure
$t_1[\ {}^{r_1}\sigma_1\ ]$	$(r_1\ ,\ t_1)$
id	$(r_2\ ,\ \Gamma_2)$
${}^{r_3}\sigma_3 + A$	$(r_3\ ,\ \sigma_3)$
$\mathsf{suc}[\ q_4\ ]\ t_4$	(q_4)

Table 1: Per-function termination measures

Luckily, there is an easy solution: making id polymorphic in its Sort and instantiating with V at the call-sites enables the decrease to be tracked and termination to be correctly inferred by Agda. We present the call graph diagrammatically (inlining _↑_), in the style of [17] (Figure 1).

To justify termination, we note that along all cycles in the graph, either the Sort strictly decreases, or the Sort is preserved and some other argument (the context, substitution, or term) decreases. Following this, we can assign lexicographically-decreasing measures to each of the functions (Table 1).

In practice, we will generally require identity renamings, rather than substitutions. We define Sort-polymorphic id-poly, and then recover our original id by instantiating it at V (and using an **INLINE** pragma so Agda's termination checker cannot tell the difference).

```
id-poly : Γ ⊩[ q ] Γ        id = id-poly {q = V}
id      : Γ ⊩[ V ] Γ        {-# INLINE id #-}
```

(All this fuss with Sort-polymorphic id may be unnecessary. At a cost in performance, it is possible to extend Agda's termination checker so that our original definitions are accepted directly. See #7695.)

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 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 ($\mathsf{xs} \circ \mathsf{id} \equiv \mathsf{xs}$). It is easy because it doesn't depend on any other categorical equations.

The main lemma is the identity law for the substitution functor $[\mathsf{id}] : \mathsf{x}\,[\,\mathsf{id}\,] \equiv \mathsf{x}$. To prove the successor case, we need naturality of $\mathsf{suc}[\,\mathsf{q}\,]$ applied to a variable, which can be shown by simple induction over said variable:

$$
\begin{array}{l}
{}^{+}\text{-nat}[]\mathsf{v} : \mathsf{i}\,[\,\mathsf{xs}\,{}^{+}\,\mathsf{A}\,] \equiv \mathsf{suc}[\,\mathsf{q}\,]\,(\mathsf{i}\,[\,\mathsf{xs}\,])\,\mathsf{A} \\
{}^{+}\text{-nat}[]\mathsf{v}\,\{\mathsf{i} = \mathsf{zero}\}\quad\{\mathsf{xs} = \mathsf{xs}\,,\mathsf{x}\} = \mathsf{refl} \\
{}^{+}\text{-nat}[]\mathsf{v}\,\{\mathsf{i} = \mathsf{suc}\,\mathsf{j}\,\mathsf{A}\}\,\{\mathsf{xs} = \mathsf{xs}\,,\mathsf{x}\} = {}^{+}\text{-nat}[]\mathsf{v}\,\{\mathsf{i} = \mathsf{j}\}
\end{array}
$$

The identity law is now easily provable by structural induction:

$$
\begin{array}{ll}
[\mathsf{id}]\,\{\mathsf{x} = \mathsf{zero}\} \quad = \mathsf{refl} & [\mathsf{id}]\,\{\mathsf{x} = \grave{}\,\mathsf{i}\} \quad = \\
[\mathsf{id}]\,\{\mathsf{x} = \mathsf{suc}\,\mathsf{i}\,\mathsf{A}\} = & \quad \mathsf{cong}\,\grave{}\_\,([\mathsf{id}]\,\{\mathsf{x} = \mathsf{i}\}) \\
\quad \mathsf{i}\,[\,\mathsf{id}\,{}^{+}\,\mathsf{A}\,] \equiv\langle\, {}^{+}\text{-nat}[]\mathsf{v}\,\{\mathsf{i} = \mathsf{i}\} \,\rangle & [\mathsf{id}]\,\{\mathsf{x} = \mathsf{t} \cdot \mathsf{u}\} = \\
\quad \mathsf{suc}\,(\mathsf{i}\,[\,\mathsf{id}\,])\,\mathsf{A} & \quad \mathsf{cong}_2\,\_\cdot\_\,([\mathsf{id}]\,\{\mathsf{x} = \mathsf{t}\})\,([\mathsf{id}]\,\{\mathsf{x} = \mathsf{u}\}) \\
\quad \equiv\langle\, \mathsf{cong}\,(\lambda\,\mathsf{j} \to \mathsf{suc}\,\mathsf{j}\,\mathsf{A})\,([\mathsf{id}]\,\{\mathsf{x} = \mathsf{i}\}) \,\rangle & [\mathsf{id}]\,\{\mathsf{x} = \lambda\,\mathsf{t}\} \quad = \\
\quad \mathsf{suc}\,\mathsf{i}\,\mathsf{A} \quad \blacksquare & \quad \mathsf{cong}\,\lambda\_\,([\mathsf{id}]\,\{\mathsf{x} = \mathsf{t}\})
\end{array}
$$

Note that the $\lambda\_$ case is easy here: we need the law to hold for $\mathsf{t} : \Gamma\,,\mathsf{A} \vdash [\,\mathsf{T}\,]\,\mathsf{B}$, but this is still covered by the inductive hypothesis because $\mathsf{id}\,\{\Gamma = \Gamma\,,\mathsf{A}\} = \mathsf{id} \uparrow \mathsf{A}$.

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, exploiting Agda's flexible syntax. Here $\mathsf{e} \equiv\langle\, \mathsf{p} \,\rangle\, \mathsf{e}'$ means that p is a proof of $\mathsf{e} \equiv \mathsf{e}'$. Later we will also use the special case $\mathsf{e} \equiv\langle\rangle\, \mathsf{e}'$ which means that e and e' are definitionally equal (this corresponds to $\mathsf{e} \equiv\langle\, \mathsf{refl} \,\rangle\, \mathsf{e}'$ and is just used to make the proof more readable). The proof is terminated with $\blacksquare$ which inserts refl. We also make heavy use of congruence $\mathsf{cong}\,\mathsf{f} : \mathsf{a} \equiv \mathsf{b} \to \mathsf{f}\,\mathsf{a} \equiv \mathsf{f}\,\mathsf{b}$ and a version for binary functions $\mathsf{cong}_2\,\mathsf{g} : \mathsf{a} \equiv \mathsf{b} \to \mathsf{c} \equiv \mathsf{d} \to \mathsf{g}\,\mathsf{a}\,\mathsf{c} \equiv \mathsf{g}\,\mathsf{b}\,\mathsf{d}$.

The category law $\circ\mathsf{id} : \mathsf{xs} \circ \mathsf{id} \equiv \mathsf{xs}$ is now simply a fold of the functor law ($[\mathsf{id}]$).

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: $[\circ] : \mathsf{x}\,[\,\mathsf{xs} \circ \mathsf{ys}\,] \equiv \mathsf{x}\,[\,\mathsf{xs}\,]\,[\,\mathsf{ys}\,]$. Even stating this signature requires (definitional) associativity of $\_ \sqcup \_$, since the left hand side has type $\Delta \vdash [\,\mathsf{q} \sqcup (\mathsf{r} \sqcup \mathsf{s})\,]\,\mathsf{A}$ while the right hand side has type $\Delta \vdash [\,(\mathsf{q} \sqcup \mathsf{r}) \sqcup \mathsf{s}\,]\,\mathsf{A}$. Fortunately, we obtain this via the $\sqcup\sqcup$ rewrite rule, but alternatively we would have to insert a transport using subst.

Of course, we must also state the left-identity law $\mathsf{id}\circ : \mathsf{id} \circ \mathsf{xs} \equiv \mathsf{xs}$. Similarly to id, Agda will not accept a direct implementation of $\mathsf{id}\circ$ 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 substitution (i.e. at sort T), its sort must be lifted on the RHS (e.g. by extending the $\mathsf{tm}\sqsubset$ functor to lists of terms) to obey $\_ \sqcup \_$.

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 5.2). Therefore, we instead add a "dummy" Sort argument (i.e. $\mathsf{id}\circ' : \mathsf{Sort} \to \mathsf{id} \circ \mathsf{xs} \equiv \mathsf{xs}$) to track the size decrease (such that we can eventually just use $\mathsf{id}\circ = \mathsf{id}\circ'\ \mathsf{V}$).

(Perhaps surprisingly, this "dummy" argument does not even need to be of type Sort to satisfy Agda here. More discussion on this trick can be found at Agda issue #7693, but in summary: (i) Agda considers all base constructors (constructors with no parameters) to be of minimal size structurally, so their presence can track size preservation of other base-constructor arguments across function calls. (ii) It turns out that a strict decrease in Sort is not necessary everywhere for termination: note that the context also gets structurally smaller in the call to $\_^+\_$ from id.)

To prove $\mathsf{id}\circ'$, we need the β-law for $\_^+\_$, $\mathsf{xs}\ ^+\ \mathsf{A} \circ (\mathsf{ys}\ ,\ \mathsf{x}) \equiv \mathsf{xs} \circ \mathsf{ys}$, which can be shown with a fold over a corresponding property for $\mathsf{suc}[\_]$.

```
suc[] : (suc[ q ] x _) [ ys , y ] ≡ x [ ys ]
suc[] {q = V} = refl
suc[] {q = T} {x = x} {ys = ys} {y = y} =
  (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 ] ■
```

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

id∘' {xs = ε}      _ = refl
id∘' {xs = xs , x} _ = cong₂ _,_
  (id ^ _ ∘ (xs , x) ≡⟨ ^∘ {xs = id} ⟩
  id ∘ xs            ≡⟨ id∘ ⟩
  xs                 ■)
  refl
```

One may note that $^+\circ$ relies on itself indirectly via suc[]. Like with the substitution operations, termination is justified here by the Sort decreasing.

4.3 Associativity

We finally get to the proof of the second functor law ($[\circ] : \mathsf{x}\ [\ \mathsf{xs} \circ \mathsf{ys}\ ] \equiv \mathsf{x}\ [\ \mathsf{xs}\ ]\ [\ \mathsf{ys}\ ]$), the main lemma for associativity. The main obstacle is that for the $\lambda\_$ case; we need the second functor law for context extension: $\uparrow\circ : (\mathsf{xs} \circ \mathsf{ys}) \uparrow \mathsf{A} \equiv (\mathsf{xs} \uparrow \mathsf{A}) \circ (\mathsf{ys} \uparrow \mathsf{A})$.

To verify the variable case we also need that $\mathsf{tm}\sqsubseteq$ commutes with substitution, $\mathsf{tm}[] : \mathsf{tm} \sqsubseteq \sqsubseteq \mathsf{t}\ (\mathsf{x}\ [\ \mathsf{xs}\ ]) \equiv (\mathsf{tm} \sqsubseteq \sqsubseteq \mathsf{t}\ \mathsf{x})\ [\ \mathsf{xs}\ ]$, which is easy to prove by case analysis.

We are now ready to prove $[\circ]$ 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 ↑ _ ] ■)
```

Associativity $\circ\circ : \mathsf{xs} \circ (\mathsf{ys} \circ \mathsf{zs}) \equiv (\mathsf{xs} \circ \mathsf{ys}) \circ \mathsf{zs}$ can be proven by folding $[\circ]$ over substitutions.

However, we are not done yet. We still need to prove the second functor law for $\_\uparrow\_$ ($\uparrow\circ$). It turns out that this depends on the naturality of weakening $^+\!-\ \mathsf{nat} \circ : \mathsf{xs} \circ (\mathsf{ys}\ ^+\ \mathsf{A}) \equiv (\mathsf{xs} \circ \mathsf{ys})\ ^+\ \mathsf{A}$, which unsurprisingly must be shown by establishing a corresponding property for substitutions:

$^+$-nat[] : x [xs $^+$ A] ≡ suc[_] (x [xs]) A. The case q = V is just the naturality for variables which we have already proven ($^+$-nat[]v). The case for q = T is more interesting and relies again on [∘] and ∘id:

```
+-nat[] {q = T} {A = A} {x = x} {xs = 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 ] ■
```

It also turns out we need zero[] : zero[q] [xs , x] ≡ tm⊑ (⊑⊔r {q = q}) x, the β-law for zero[_], which holds definitionally in the case for either Sort. And we 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].

Finally, we have all the ingredients to prove the second functor law ↑∘:

```
↑∘ {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 [13]. We summarise 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).

That is, 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.

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 these. 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). We postulate the existence of this QIIT in Agda, with the associated β-laws implemented with rewrite rules (alternatively, we could use a truncated Cubical Agda HIT, but Cubical Agda still

lacks essential automation, e.g. it does not integrate no-confusion properties into pattern matching). 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 : Set_1.

For the contents, we begin with the category of contexts, using the same naming conventions as introduced previously:

$$\begin{aligned}
&\mathsf{Con} &&: \mathsf{Set}\\
&_\Vdash_ &&: \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{id} &&: \Gamma \Vdash \Gamma\\
&_\circ_ &&: \Delta \Vdash \Theta \to \Gamma \Vdash \Delta \to \Gamma \Vdash \Theta
\end{aligned}$$

$$\begin{aligned}
&\mathsf{id}\circ &&: \mathsf{id} \circ \delta \equiv \delta\\
&\circ\mathsf{id} &&: \delta \circ \mathsf{id} \equiv \delta\\
&\circ\circ &&: (\xi \circ \theta) \circ \delta \equiv \xi \circ (\theta \circ \delta)
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{Ty} &&: \mathsf{Set}\\
&_\vdash_ &&: \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set}\\
&_[_] &&: \Gamma \vdash \mathsf{A} \to \Delta \Vdash \Gamma \to \Delta \vdash \mathsf{A}
\end{aligned}$$

$$\begin{aligned}
&[\mathsf{id}] &&: (\mathsf{t}\,[\,\mathsf{id}\,]) \equiv \mathsf{t}\\
&[\circ] &&: \mathsf{t}\,[\,\theta\,]\,[\,\delta\,] \equiv \mathsf{t}\,[\,\theta \circ \delta\,]
\end{aligned}$$

The category of contexts has a terminal object (the empty context), and context extension resembles categorical products but mixing contexts and types:

$$\begin{aligned}
&\bullet &&: \mathsf{Con}\\
&\varepsilon &&: \Gamma \Vdash \bullet\\
&_\triangleright_ &&: \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Con}\\
&_,_ &&: \Gamma \Vdash \Delta \to \Gamma \vdash \mathsf{A} \to \Gamma \Vdash (\Delta \triangleright \mathsf{A})\\
&\pi_0 &&: \Gamma \Vdash (\Delta \triangleright \mathsf{A}) \to \Gamma \Vdash \Delta\\
&\pi_1 &&: \Gamma \Vdash (\Delta \triangleright \mathsf{A}) \to \Gamma \vdash \mathsf{A}
\end{aligned}$$

$$\begin{aligned}
&\bullet\!-\eta &&: \delta \equiv \varepsilon\\
&\triangleright\!-\beta_0 &&: \pi_0\,(\delta\,,\mathsf{t}) \equiv \delta\\
&\triangleright\!-\beta_1 &&: \pi_1\,(\delta\,,\mathsf{t}) \equiv \mathsf{t}\\
&\triangleright\!-\eta &&: (\pi_0\,\delta\,,\pi_1\,\delta) \equiv \delta\\
&\pi_0\circ &&: \pi_0\,(\theta \circ \delta) \equiv \pi_0\,\theta \circ \delta\\
&\pi_1\circ &&: \pi_1\,(\theta \circ \delta) \equiv (\pi_1\,\theta)\,[\,\delta\,]
\end{aligned}$$

We can define the morphism part of the context extension functor as before, $\delta \uparrow \mathsf{A} = (\delta \circ (\pi_0\,\mathsf{id}))\,,\pi_1\,\mathsf{id}$. 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:

$$\begin{aligned}
&\mathsf{o} &&: \mathsf{Ty}\\
&_\Rightarrow_ &&: \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty}\\
&_\cdot_ &&: \Gamma \vdash \mathsf{A} \Rightarrow \mathsf{B} \to \Gamma \vdash \mathsf{A} \to \Gamma \vdash \mathsf{B}
\end{aligned}$$

$$\begin{aligned}
&\lambda_ &&: \Gamma \triangleright \mathsf{A} \vdash \mathsf{B} \to \Gamma \vdash \mathsf{A} \Rightarrow \mathsf{B}\\
&\cdot[] &&: (\mathsf{t} \cdot \mathsf{u})\,[\,\delta\,] \equiv (\mathsf{t}\,[\,\delta\,]) \cdot (\mathsf{u}\,[\,\delta\,])\\
&\lambda[] &&: (\lambda\,\mathsf{t})\,[\,\delta\,] \equiv \lambda\,(\mathsf{t}\,[\,\delta \uparrow _\,])
\end{aligned}$$

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 $\_\vdash[\_]\_/\_\Vdash[\_]\_$. This will be more-or-less enough to implement the "normalisation" direction of our initial CwF $\simeq$ recursive substitution syntax isomorphism.

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

Our first non-trivial decision is which type family to interpret substitutions into. In our first attempt, we tried to pair renamings/substitutions with their sorts to stay polymorphic, is-cwf .CwF._ ⊩ _ = Σ Sort (Δ ⊩[_] Γ). Unfortunately, this approach quickly breaks. The •−η CwF law forces us to provide a unique morphism to the terminal context (i.e. a unique weakening from the empty context); Σ Sort (Δ ⊩[_] Γ) is simply too flexible here, allowing both V , ε and T , ε.
Therefore, we instead fix the sort to T.

```
is-cwf .CwF._ ⊩ _ = _ ⊩[ T ]_        is-cwf .CwF.•−η {δ = ε} = refl
is-cwf .CwF.•     = •                is-cwf .CwF._∘_         = _∘_
is-cwf .CwF.ε     = ε                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 renamings/substitutions with a fold: tm*⊑ : q ⊑ s → Γ ⊩[q] Δ → Γ ⊩[s] Δ, and prove various lemmas about how tm*⊑ coercions can be lifted outside of our substitution operators:

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

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 folding over substitutions using 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 ■
```

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 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} =                is-cwf .CwF.∘id {δ = δ} =
  tm*⊑ v⊑t id ∘ δ ≡⟨ ⊑∘ ⟩                  δ ∘ tm*⊑ v⊑t id ≡⟨ ∘⊑ ⟩
  id ∘ δ         ≡⟨ id∘ ⟩                  δ ∘ id         ≡⟨ ∘id ⟩
  δ ■                                      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.[id] {t = t} =
  t [ tm*⊑ v⊑t id ] ≡⟨ t[⊑] {t = t} ⟩
  t [ id ]          ≡⟨ [id] ⟩
  t                 ■

is-cwf .CwF._⊢_ = _⊢[ T ]_
is-cwf .CwF._[_] = _[_]
```

We now define projections π₀ (δ , t) = δ and π₁ (δ , t) = t and ▷−β₀, ▷−β₁, ▷−η, π₀∘ and π₁∘ all hold by definition (at least, after matching on the guaranteed-non-empty substitution).

Finally, we can deal with the cases specific to simply typed λ-calculus. ·[] also holds by definition, but the β-rule for substitutions applied to lambdas requires a bit of equational reasoning due to differing implementations of _↑_.

$$
\begin{aligned}
&\mathsf{is\text{-}cwf}\,.\mathsf{CwF}.\lambda[]\,\{\mathsf{A}=\mathsf{A}\}\,\{\mathsf{t}=\mathsf{x}\}\,\{\delta=\mathsf{ys}\} = \\
&\quad \lambda\,\mathsf{x}\,[\,\mathsf{ys}\uparrow\mathsf{A}\,] && \equiv\langle\,\mathsf{cong}\,(\lambda\,\rho\to\lambda\,\mathsf{x}\,[\,\rho\uparrow\mathsf{A}\,])\,(\mathsf{sym}\circ\mathsf{id})\,\rangle \\
&\quad \lambda\,\mathsf{x}\,[\,(\mathsf{ys}\circ\mathsf{id})\uparrow\mathsf{A}\,] && \equiv\langle\,\mathsf{cong}\,(\lambda\,\rho\to\lambda\,\mathsf{x}\,[\,\rho\,,`\,\mathsf{zero}\,])\,(\mathsf{sym}\,{}^{+}\!-\mathsf{nat}\circ)\,\rangle \\
&\quad \lambda\,\mathsf{x}\,[\,\mathsf{ys}\circ\mathsf{id}\,{}^{+}\,\mathsf{A}\,,`\,\mathsf{zero}\,] && \equiv\langle\,\mathsf{cong}\,(\lambda\,\rho\to\lambda\,\mathsf{x}\,[\,\rho\,,`\,\mathsf{zero}\,])\,(\mathsf{sym}\,(\circ\sqsubseteq\,\{\mathsf{ys}=\mathsf{id}\,{}^{+}\,\_\}))\,\rangle \\
&\quad \lambda\,\mathsf{x}\,[\,\mathsf{ys}\circ\mathsf{tm}^{*}\sqsubseteq\,\mathsf{v}\sqsubseteq\mathsf{t}\,(\mathsf{id}\,{}^{+}\,\mathsf{A})\,,`\,\mathsf{zero}\,] && \blacksquare
\end{aligned}
$$

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 can reuse our existing datatypes for contexts and types because in STLC there are no non-trivial equations on these components.

To avoid name clashes between our existing syntax and the initial CwF constructors, we annotate every ICwF constructor with $^{\mathrm{I}}$. e.g. $\_\vdash^{\mathrm{I}}\_ : \mathsf{Con}\to\mathsf{Ty}\to\mathsf{Set}$, $\mathsf{id}^{\mathrm{I}} : \Gamma\Vdash^{\mathrm{I}}\Gamma$ etc.

We state the eliminator for the initial CwF assuming appropriate $\mathsf{Motive} : \mathsf{Set}_1$ and $\mathsf{Methods} : \mathsf{Motive}\to\mathsf{Set}_1$ records as in [7]. Again to avoid name clashes, we annotate fields of these records (corresponding to how each type/constructor is eliminated) with $^{\mathrm{M}}$.

$$
\begin{aligned}
&\mathsf{elim\text{-}con} : \forall\,\Gamma\to\mathsf{Con}^{\mathrm{M}}\,\Gamma && \mathsf{elim\text{-}cwf} : \forall\,\mathsf{t}^{\mathrm{I}}\to\mathsf{Tm}^{\mathrm{M}}\,(\mathsf{elim\text{-}con}\,\Gamma)\,(\mathsf{elim\text{-}ty}\,\mathsf{A})\,\mathsf{t}^{\mathrm{I}} \\
&\mathsf{elim\text{-}ty} : \forall\,\mathsf{A}\to\mathsf{Ty}^{\mathrm{M}}\,\mathsf{A} && \mathsf{elim\text{-}cwf}^{*} : \forall\,\delta^{\mathrm{I}}\to\mathsf{Tms}^{\mathrm{M}}\,(\mathsf{elim\text{-}con}\,\Delta)\,(\mathsf{elim\text{-}con}\,\Gamma)\,\delta^{\mathrm{I}}
\end{aligned}
$$

To state the dependent equations in Methods between outputs of the eliminator, enforcing congruence of equality (e.g. the functor law, which asks for $\mathsf{t}^{\mathrm{M}}\,[\,\sigma^{\mathrm{M}}\,]^{\mathrm{M}}\,[\,\delta^{\mathrm{M}}\,]^{\mathrm{M}}$ and $\mathsf{t}^{\mathrm{M}}\,[\,\sigma^{\mathrm{M}}\circ^{\mathrm{M}}\delta^{\mathrm{M}}\,]^{\mathrm{M}}$ to be equated) we need dependent identity types $\_\equiv[\_]\equiv\_ : \mathsf{A}\to\mathsf{A}\equiv\mathsf{B}\to\mathsf{B}\to\mathsf{Set}\,\ell$. We can define these simply by matching on the identity between A and B, $\mathsf{x}\equiv[\,\mathsf{refl}\,]\equiv\mathsf{y} = \mathsf{x}\equiv\mathsf{y}$.

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. To achieve this, we define $\mathsf{cwf\text{-}to\text{-}motive} : \mathsf{CwF\text{-}simple}\to\mathsf{Motive}$ and $\mathsf{cwf\text{-}to\text{-}methods} : \mathsf{CwF\text{-}simple}\to\mathsf{Methods}$, which simply project out the relevant fields, and then implement e.g. $\mathsf{rec\text{-}cwf} = \mathsf{elim\text{-}cwf}\,\mathsf{cwf\text{-}to\text{-}methods}$.

The one extra ingredient we need to make this work out neatly is to introduce a new reduction for cong, $\mathsf{cong}\,(\lambda\,\_\to\mathsf{x})\,\mathsf{p}\equiv\mathsf{refl}$, via an Agda rewrite rule (this identity also holds natively in Cubical). This enables the no-longer-dependent $_\equiv[_]\equiv_$s to collapse to $_\equiv_$s automatically.
Normalisation into our substitution normal forms can now be achieved by with:

$$
\begin{aligned}
&\mathsf{norm} : \Gamma\vdash^{\mathrm{I}}\mathsf{A}\to\mathsf{rec\text{-}con}\,\mathsf{is\text{-}cwf}\,\Gamma\vdash[\,\mathsf{T}\,]\,\mathsf{rec\text{-}ty}\,\mathsf{is\text{-}cwf}\,\mathsf{A} \\
&\mathsf{norm} = \mathsf{rec\text{-}cwf}\,\mathsf{is\text{-}cwf}
\end{aligned}
$$

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 $\Gamma\vdash^{\mathrm{I}}\mathsf{A}\to\Gamma\vdash[\,\mathsf{T}\,]\,\mathsf{A}$ and, conveniently, rewrite rules ($\mathsf{rec\text{-}con}\,\mathsf{is\text{-}cwf}\,\Gamma\equiv\Gamma$ and $\mathsf{rec\text{-}ty}\,\mathsf{is\text{-}cwf}\,\mathsf{A}\equiv\mathsf{A}$) can get us there!

$$
\begin{aligned}
&\mathsf{norm} : \Gamma\vdash^{\mathrm{I}}\mathsf{A}\to\Gamma\vdash[\,\mathsf{T}\,]\,\mathsf{A} && \mathsf{norm}^{*} : \Delta\Vdash^{\mathrm{I}}\Gamma\to\Delta\Vdash[\,\mathsf{T}\,]\,\Gamma \\
&\mathsf{norm} = \mathsf{rec\text{-}cwf}\,\mathsf{is\text{-}cwf} && \mathsf{norm}^{*} = \mathsf{rec\text{-}cwf}^{*}\,\mathsf{is\text{-}cwf}
\end{aligned}
$$

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

$$
\begin{aligned}
&\ulcorner\_\urcorner : \Gamma\vdash[\,\mathsf{q}\,]\,\mathsf{A}\to\Gamma\vdash^{\mathrm{I}}\mathsf{A} && \ulcorner\,\mathsf{t}\cdot\mathsf{u}\,\urcorner = \ulcorner\,\mathsf{t}\,\urcorner\,.^{\mathrm{I}}\,\ulcorner\,\mathsf{u}\,\urcorner \\
&\ulcorner\_\urcorner^{*} : \Delta\Vdash[\,\mathsf{q}\,]\,\Gamma\to\Delta\Vdash^{\mathrm{I}}\Gamma && \ulcorner\,\lambda\,\mathsf{t}\,\urcorner = \lambda^{\mathrm{I}}\,\ulcorner\,\mathsf{t}\,\urcorner \\
&\ulcorner\,\mathsf{zero}\,\urcorner = \mathsf{zero}^{\mathrm{I}} && \ulcorner\,\varepsilon\,\urcorner^{*} = \varepsilon^{\mathrm{I}} \\
&\ulcorner\,\mathsf{suc}\,\mathsf{i}\,\mathsf{B}\,\urcorner = \mathsf{suc}^{\mathrm{I}}\,\ulcorner\,\mathsf{i}\,\urcorner\,\mathsf{B} && \ulcorner\,\delta\,,\mathsf{x}\,\urcorner^{*} = \ulcorner\,\delta\,\urcorner^{*}\,,^{\mathrm{I}}\,\ulcorner\,\mathsf{x}\,\urcorner \\
&\ulcorner\,`\,\mathsf{i}\,\urcorner = \ulcorner\,\mathsf{i}\,\urcorner
\end{aligned}
$$

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 $\ulcorner\_\urcorner$ are mutual inverses - i.e. stability ($\mathsf{norm}\ulcorner\, \mathsf{t}\,\urcorner \equiv \mathsf{t}$) and completeness ($\ulcorner\, \mathsf{norm}\,\mathsf{t}\,\urcorner \equiv \mathsf{t}$).

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 slightly generalise the inductive hypothesis, 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.

$$
\begin{aligned}
&\mathsf{stab} : \mathsf{norm}\ulcorner\, \mathsf{x}\,\urcorner \equiv \mathsf{tm}\sqsubseteq\sqsubseteq\mathsf{t}\,\mathsf{x}\\
&\mathsf{stab}\,\{\mathsf{x} = \mathsf{zero}\} \quad = \mathsf{refl}\\
&\mathsf{stab}\,\{\mathsf{x} = \mathsf{suc}\,\mathsf{i}\,\mathsf{B}\} =\\
&\quad \mathsf{norm}\ulcorner\, \mathsf{i}\,\urcorner[\,\mathsf{tm}^*\sqsubseteq\ \mathsf{v}\sqsubseteq\mathsf{t}\,(\mathsf{id}^+\,\mathsf{B})\,] \equiv\langle\, \mathsf{t}[\sqsubseteq]\,\{\mathsf{t} = \mathsf{norm}\ulcorner\, \mathsf{i}\,\urcorner\}\,\rangle\\
&\quad \mathsf{norm}\ulcorner\, \mathsf{i}\,\urcorner[\,\mathsf{id}^+\,\mathsf{B}\,] \equiv\langle\, \mathsf{cong}\,(\lambda\,\mathsf{j}\,\to\,\mathsf{suc}[\,\_\,]\,\mathsf{j}\,\mathsf{B})\,(\mathsf{stab}\,\{\mathsf{x} = \mathsf{i}\})\,\rangle\\
&\quad \grave{}\,\mathsf{i}\,[\,\mathsf{id}^+\,\mathsf{B}\,] \equiv\langle\, \mathsf{cong}\,\grave{}\_\,\mathsf{suc}[\mathsf{id}^+]\,\rangle\\
&\quad \grave{}\,\mathsf{suc}\,\mathsf{i}\,\mathsf{B}\ \blacksquare\\
&\mathsf{stab}\,\{\mathsf{x} = \grave{}\,\mathsf{i}\} \quad = \mathsf{stab}\,\{\mathsf{x} = \mathsf{i}\}\\
&\mathsf{stab}\,\{\mathsf{x} = \mathsf{t}\cdot\mathsf{u}\} \quad = \mathsf{cong}_2\,\_\cdot\_\,(\mathsf{stab}\,\{\mathsf{x} = \mathsf{t}\})\,(\mathsf{stab}\,\{\mathsf{x} = \mathsf{u}\})\\
&\mathsf{stab}\,\{\mathsf{x} = \lambda\,\mathsf{t}\} \quad = \mathsf{cong}\,\lambda\_\,(\mathsf{stab}\,\{\mathsf{x} = \mathsf{t}\})
\end{aligned}
$$

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

$$\mathsf{compl}\text{-}\mathbb{M} : \mathsf{Motive}$$

$$
\begin{aligned}
&\mathsf{compl}\text{-}\mathbb{M}.\mathsf{Tm}^{\mathrm{M}}\ \_\ \_\ \mathsf{t}^{\mathrm{I}} \quad = \ulcorner\, \mathsf{norm}\,\mathsf{t}^{\mathrm{I}}\,\urcorner \equiv \mathsf{t}^{\mathrm{I}} && \mathsf{compl}\text{-}\mathbb{M}.\mathsf{Con}^{\mathrm{M}}\ \_ = \top\\
&\mathsf{compl}\text{-}\mathbb{M}.\mathsf{Tms}^{\mathrm{M}}\ \_\ \_\ \delta^{\mathrm{I}} = \ulcorner\, \mathsf{norm}^*\,\delta^{\mathrm{I}}\,\urcorner^* \equiv \delta^{\mathrm{I}} && \mathsf{compl}\text{-}\mathbb{M}.\mathsf{Ty}^{\mathrm{M}}\ \_ \quad = \top
\end{aligned}
$$

To show these identities, we need to prove that our various recursively defined syntax operations are preserved by $\ulcorner\_\urcorner$.

Preservation of $\mathsf{zero}[\_]$, $\ulcorner\mathsf{zero}\urcorner : \ulcorner\, \mathsf{zero}[\,\mathsf{q}\,]\,\urcorner \equiv \mathsf{zero}^{\mathrm{I}}$ reduces to reflexivity after splitting on the sort.

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. $\triangleright\text{-}\beta_0^{\mathrm{I}}$).

Preservation proofs for $\_[\_]$, $\_\uparrow\_$, $\_^+\_$, id and $\mathsf{suc}[\_]$ 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 $\ulcorner\mathsf{id}\urcorner$ to keep Agda's termination checker happy.

$$
\begin{aligned}
&\ulcorner[]\urcorner : \ulcorner\, \mathsf{x}\,[\,\mathsf{ys}\,]\,\urcorner \equiv \ulcorner\, \mathsf{x}\,\urcorner[\,\ulcorner\, \mathsf{ys}\,\urcorner^*\,]^{\mathrm{I}}\\
&\ulcorner\uparrow\urcorner : \ulcorner\, \mathsf{xs}\uparrow\mathsf{A}\,\urcorner^* \equiv \ulcorner\, \mathsf{xs}\,\urcorner^*\uparrow^{\mathrm{I}}\mathsf{A}\\
&\ulcorner{+}\urcorner : \ulcorner\, \mathsf{xs}^+\mathsf{A}\,\urcorner^* \equiv \ulcorner\, \mathsf{xs}\,\urcorner^*\circ^{\mathrm{I}}\mathsf{wk}^{\mathrm{I}}\\
&\ulcorner\mathsf{id}\urcorner : \ulcorner\, \mathsf{id}\,\{\Gamma = \Gamma\}\,\urcorner^* \equiv \mathsf{id}^{\mathrm{I}}
\end{aligned}
\qquad
\begin{aligned}
&\ulcorner\mathsf{suc}\urcorner : \ulcorner\, \mathsf{suc}[\,\mathsf{q}\,]\,\mathsf{x}\,\mathsf{B}\,\urcorner \equiv \ulcorner\, \mathsf{x}\,\urcorner[\,\mathsf{wk}^{\mathrm{I}}\,]^{\mathrm{I}}\\
&\ulcorner\mathsf{id}\urcorner' : \mathsf{Sort}\to\ulcorner\, \mathsf{id}\,\{\Gamma = \Gamma\}\,\urcorner^* \equiv \mathsf{id}^{\mathrm{I}}\\
&\ulcorner\mathsf{id}\urcorner = \ulcorner\mathsf{id}\urcorner'\,\mathsf{V}
\end{aligned}
$$

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

$$\mathsf{zero}[]^{\mathrm{I}} : \mathsf{zero}^{\mathrm{I}}\,[\ \delta^{\mathrm{I}}\ ,^{\mathrm{I}}\ \mathsf{t}^{\mathrm{I}}\ ]^{\mathrm{I}} \equiv \mathsf{t}^{\mathrm{I}}$$
$$\mathsf{zero}[]^{\mathrm{I}}\,\{\delta^{\mathrm{I}} = \delta^{\mathrm{I}}\}\,\{\mathsf{t}^{\mathrm{I}} = \mathsf{t}^{\mathrm{I}}\} =$$
$$\begin{aligned}
&\mathsf{zero}^{\mathrm{I}}\,[\ \delta^{\mathrm{I}}\ ,^{\mathrm{I}}\ \mathsf{t}^{\mathrm{I}}\ ]^{\mathrm{I}} && \equiv\langle\ \mathsf{sym}\ \pi_1 \circ^{\mathrm{I}}\ \rangle\\
&\pi_1^{\mathrm{I}}\ (\mathsf{id}^{\mathrm{I}} \circ^{\mathrm{I}} (\delta^{\mathrm{I}}\ ,^{\mathrm{I}}\ \mathsf{t}^{\mathrm{I}})) && \equiv\langle\ \mathsf{cong}\ \pi_1^{\mathrm{I}}\ \mathsf{id} \circ^{\mathrm{I}}\ \rangle\\
&\pi_1^{\mathrm{I}}\ (\delta^{\mathrm{I}}\ ,^{\mathrm{I}}\ \mathsf{t}^{\mathrm{I}}) && \equiv\langle\ \triangleright\!-\!\beta_1^{\mathrm{I}}\ \rangle\\
&\mathsf{t}^{\mathrm{I}} && \blacksquare
\end{aligned}$$

$$\mathsf{suc}[]^{\mathrm{I}} : \mathsf{suc}^{\mathrm{I}}\ \mathsf{t}^{\mathrm{I}}\ \mathsf{B}\ [\ \delta^{\mathrm{I}}\ ,^{\mathrm{I}}\ \mathsf{u}^{\mathrm{I}}\ ]^{\mathrm{I}} \equiv \mathsf{t}^{\mathrm{I}}\ [\ \delta^{\mathrm{I}}\ ]^{\mathrm{I}}$$
$$\mathsf{suc}[]^{\mathrm{I}} = \ldots$$

$$,\circ^{\mathrm{I}} : (\delta^{\mathrm{I}}\ ,^{\mathrm{I}}\ \mathsf{t}^{\mathrm{I}}) \circ^{\mathrm{I}} \sigma^{\mathrm{I}} \equiv (\delta^{\mathrm{I}} \circ^{\mathrm{I}} \sigma^{\mathrm{I}})\ ,^{\mathrm{I}}\ (\mathsf{t}^{\mathrm{I}}\ [\ \sigma^{\mathrm{I}}\ ]^{\mathrm{I}})$$
$$,\circ^{\mathrm{I}} = \ldots$$

We also need a couple lemmas about how $\ulcorner\_\urcorner$ treats terms of different sorts identically: $\ulcorner\sqsubseteq\urcorner : \forall\,\{\mathsf{x} : \Gamma \vdash [\ \mathsf{q}\ ]\ \mathsf{A}\} \to \ulcorner\ \mathsf{tm}\sqsubseteq\sqsubseteq\mathsf{t}\ \mathsf{x}\ \urcorner \equiv \ulcorner\ \mathsf{x}\ \urcorner$ and $\ulcorner\sqsubseteq\urcorner\mathsf{*} : \ulcorner\ \mathsf{tm}\mathsf{*}\sqsubseteq\sqsubseteq\mathsf{t}\ \mathsf{xs}\ \urcorner\mathsf{*} \equiv \ulcorner\ \mathsf{xs}\ \urcorner\mathsf{*}$.

We can now proceed with the preservation proofs. There are quite a few cases to cover, so for brevity we elide the proofs of $\ulcorner[]\urcorner$ and $\ulcorner\mathsf{suc}\urcorner$.

$$\ulcorner\uparrow\urcorner\,\{\mathsf{q} = \mathsf{q}\} = \mathsf{cong}_2\ \_,^{\mathrm{I}}\_\ \ulcorner+\urcorner\ (\ulcorner\mathsf{zero}\urcorner\,\{\mathsf{q} = \mathsf{q}\})$$

$$\ulcorner+\urcorner\,\{\mathsf{xs} = \varepsilon\} = \mathsf{sym}\ \bullet\!-\!\eta^{\mathrm{I}}$$
$$\ulcorner+\urcorner\,\{\mathsf{xs} = \mathsf{xs}\,,\,\mathsf{x}\}\,\{\mathsf{A} = \mathsf{A}\} =$$
$$\begin{aligned}
&\ulcorner\ \mathsf{xs}\ ^{+}\ \mathsf{A}\ \urcorner\mathsf{*}\ ,^{\mathrm{I}}\ \ulcorner\ \mathsf{suc}[\ _\ ]\ \mathsf{x}\ \mathsf{A}\ \urcorner\\
&\equiv\langle\ \mathsf{cong}_2\ _,^{\mathrm{I}}_\ \ulcorner+\urcorner\ (\ulcorner\mathsf{suc}\urcorner\,\{\mathsf{x} = \mathsf{x}\})\ \rangle\\
&(\ulcorner\ \mathsf{xs}\ \urcorner\mathsf{*} \circ^{\mathrm{I}} \mathsf{wk}^{\mathrm{I}})\ ,^{\mathrm{I}}\ (\ulcorner\ \mathsf{x}\ \urcorner\ [\ \mathsf{wk}^{\mathrm{I}}\ ]^{\mathrm{I}})\\
&\equiv\langle\ \mathsf{sym}\ ,\circ^{\mathrm{I}}\ \rangle\\
&(\ulcorner\ \mathsf{xs}\ \urcorner\mathsf{*}\ ,^{\mathrm{I}}\ \ulcorner\ \mathsf{x}\ \urcorner)\circ^{\mathrm{I}}\mathsf{wk}^{\mathrm{I}}\ \blacksquare
\end{aligned}$$

$$\ulcorner\mathsf{id}\urcorner'\,\{\Gamma = \bullet\}\ \_ = \mathsf{sym}\ \bullet\!-\!\eta^{\mathrm{I}}$$
$$\ulcorner\mathsf{id}\urcorner'\,\{\Gamma = \Gamma \triangleright \mathsf{A}\}\ \_ =$$
$$\begin{aligned}
&\ulcorner\ \mathsf{id}\ ^{+}\ \mathsf{A}\ \urcorner\mathsf{*}\ ,^{\mathrm{I}}\ \mathsf{zero}^{\mathrm{I}} && \equiv\langle\ \mathsf{cong}\ (_,^{\mathrm{I}}\ \mathsf{zero}^{\mathrm{I}})\ \ulcorner+\urcorner\ \rangle\\
&\ulcorner\ \mathsf{id}\ \urcorner\mathsf{*}\ \uparrow^{\mathrm{I}}\ \mathsf{A} && \equiv\langle\ \mathsf{cong}\ (_\uparrow^{\mathrm{I}}\ \mathsf{A})\ \ulcorner\mathsf{id}\urcorner\ \rangle\\
&\mathsf{id}^{\mathrm{I}}\ \uparrow^{\mathrm{I}}\ \mathsf{A} && \equiv\langle\ \mathsf{cong}\ (_,^{\mathrm{I}}\ \mathsf{zero}^{\mathrm{I}})\ \mathsf{id}\circ^{\mathrm{I}}\ \rangle\\
&\mathsf{wk}^{\mathrm{I}}\ ,^{\mathrm{I}}\ \mathsf{zero}^{\mathrm{I}} && \equiv\langle\ \triangleright\!-\!\eta^{\mathrm{I}}\ \rangle\\
&\mathsf{id}^{\mathrm{I}} && \blacksquare
\end{aligned}$$

We also prove preservation of substitution composition $\ulcorner\circ\urcorner : \ulcorner\ \mathsf{xs} \circ \mathsf{ys}\ \urcorner\mathsf{*} \equiv \ulcorner\ \mathsf{xs}\ \urcorner\mathsf{*} \circ^{\mathrm{I}} \ulcorner\ \mathsf{ys}\ \urcorner\mathsf{*}$ in similar fashion, folding $\ulcorner[]\urcorner$. The main cases of compl-$\mathbf{m}$: Methods compl-$\mathbb{M}$ can now be proved by just applying the preservation lemmas and inductive hypotheses, e.g:

$$\begin{aligned}
&\mathsf{compl}\text{-}\mathbf{m}\ .\mathsf{id}^{\mathrm{M}} = && \mathsf{compl}\text{-}\mathbf{m}\ ._\circ^{\mathrm{M}}_\,\{\sigma^{\mathrm{I}} = \sigma^{\mathrm{I}}\}\,\{\delta^{\mathrm{I}} = \delta^{\mathrm{I}}\}\,\sigma^{\mathrm{M}}\ \delta^{\mathrm{M}} =\\
&\quad\ulcorner\ \mathsf{tm}\mathsf{*}\sqsubseteq\ \mathsf{v}\sqsubseteq\mathsf{t}\ \mathsf{id}\ \urcorner\mathsf{*} \equiv\langle\ \ulcorner\sqsubseteq\urcorner\mathsf{*}\ \rangle && \quad\ulcorner\ \mathsf{norm}\mathsf{*}\ \sigma^{\mathrm{I}} \circ \mathsf{norm}\mathsf{*}\ \delta^{\mathrm{I}}\ \urcorner\mathsf{*} \equiv\langle\ \ulcorner\circ\urcorner\ \rangle\\
&\quad\ulcorner\ \mathsf{id}\ \urcorner\mathsf{*} \equiv\langle\ \ulcorner\mathsf{id}\urcorner\ \rangle && \quad\ulcorner\ \mathsf{norm}\mathsf{*}\ \sigma^{\mathrm{I}}\ \urcorner\mathsf{*} \circ^{\mathrm{I}} \ulcorner\ \mathsf{norm}\mathsf{*}\ \delta^{\mathrm{I}}\ \urcorner\mathsf{*} \equiv\langle\ \mathsf{cong}_2\ _\circ^{\mathrm{I}}_\ \sigma^{\mathrm{M}}\ \delta^{\mathrm{M}}\ \rangle\\
&\quad\mathsf{id}^{\mathrm{I}}\ \blacksquare && \quad\sigma^{\mathrm{I}} \circ^{\mathrm{I}} \delta^{\mathrm{I}}\ \blacksquare
\end{aligned}$$

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 dependent uniqueness of identity proofs (UIP)

$$\mathsf{duip} : \forall\,\{\mathsf{p} : \mathsf{x} \equiv \mathsf{y}\}\,\{\mathsf{q} : \mathsf{z} \equiv \mathsf{w}\} \to \mathsf{p} \equiv[\ \mathsf{r}\ ]\equiv \mathsf{q}$$

which enables, e.g., compl-$\mathbf{m}$ .id$\circ^{\mathrm{M}}$ = duip. Note that proving this form of UIP relies on type constructor injectivity, specifically, injectivity of $\_ \equiv \_$. We could use a weaker version, taking an additional proof of $\mathsf{x} \equiv \mathsf{z}$, but this would be clunkier to use as Agda has no hope of inferring such a proof by unification.

Completeness is now just one call to the eliminator away.

$$\mathsf{compl} : \ulcorner\ \mathsf{norm}\ \mathsf{t}^{\mathrm{I}}\ \urcorner \equiv \mathsf{t}^{\mathrm{I}}$$
$$\mathsf{compl}\,\{\mathsf{t}^{\mathrm{I}} = \mathsf{t}^{\mathrm{I}}\} = \mathsf{elim\text{-}cwf}\ \mathsf{compl}\text{-}\mathbf{m}\ \mathsf{t}^{\mathrm{I}}$$

6 Conclusions and further work

The subject of the paper is a problem which we expect many people (including ourselves) would have thought trivial. Theorem provers have made significant progress since the first POPLMark challenge [10] (which indeed focused on problems relating to binding and substitution), motivating a shifted focus (onto logical relations proofs) in newer benchmarks [2]. As it turns out, elegantly mechanising substitution still requires some care, and we spent quite some time going down alleys that didn't work (whilst getting to grips with the subtleties of Agda's termination checking).

The convenience of our solution relies on Agda's built-in support for lexicographic termination [3]. In contrast, Rocq's `Fixpoint` command merely supports structural recursion on a single argument and Lean has only raw elimination principles as primitive. Luckily, both of these proof assistants layer on additional tactics to support more natural use of non-primitive induction, making our approach somewhat transferable. Indeed, Lean can be convinced that our substitution operations terminate after specifying measures similar to those in Section 3.1, via the `termination_by` tactic.

One reviewer asked about another alternative: since we are merging $\_ \ni \_$ and $\_ \vdash \_$ 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 $\bullet : \Gamma \rhd \mathsf{A} \vdash \mathsf{A}$ and an explicit weakening operator on terms (written $\_\uparrow : \Gamma \vdash \mathsf{B} \to \Gamma \rhd \mathsf{A} \vdash \mathsf{B}$). 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 $\bullet\uparrow\uparrow \cdot \bullet\uparrow \cdot \bullet$ and $(\bullet\uparrow \cdot \bullet)\uparrow \cdot \bullet$ are equivalent, where $\bullet\uparrow\uparrow$ corresponds to the variable with de Bruijn index two. A development along these lines is explored in [24].

We see the current construction as a warmup for the definition of substitution for dependent type theory This is harder, because then the typing of the constructors actually depends on the substitution laws. Such a Münchhausian [8] construction should be possible in Agda. However, the theoretical underpinning of inductive-inductive-recursive definitions is mostly unexplored, with the exception of [16]. 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 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 (2011): *Parallel substitution as an operation for untyped de Bruijn terms*. Available at `https://www.cse.chalmers.se/~abela/html/ParallelSubstitution.html`. Agda proof.

[2] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark reloaded: Mechanizing proofs by logical relations*. *J. Funct. Program.* 29, p. e19, doi:10.1017/S0956796819000170.

[3] Andreas Abel & Thorsten Altenkirch (2002): *A predicative analysis of structural recursion*. *J. Funct. Program.* 12(1), pp. 1–41, doi:10.1017/S0956796801004191.

[4] Robin Adams (2004): *Formalized Metatheory with Terms Represented by an Indexed Family of Types*. In Jean-Christophe Filliâtre, Christine Paulin-Mohring & Benjamin Werner, editors: *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, *Lecture Notes in Computer Science* 3839, Springer, pp. 1–16, doi:10.1007/11617990_1.

[5] Guillaume Allais, James Chapman, Conor McBride & James McKinna (2017): *Type-and-scope safe programs and their proofs*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, ACM, pp. 195–207, doi:10.1145/3018610.3018613.

[6] Thorsten Altenkirch, James Chapman & Tarmo Uustalu (2015): *Monads need not be endofunctors*. *Log. Methods Comput. Sci.* 11(1), doi:10.2168/LMCS-11(1:3)2015.

[7] Thorsten Altenkirch & Ambrus Kaposi (2016): *Type theory in type theory using quotient inductive types*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 51, Association for Computing Machinery, New York, NY, USA, p. 18–29, doi:10.1145/2914770.2837638.

[8] Thorsten Altenkirch, Ambrus Kaposi, Artjoms Sinkarovs & Tamás Végh (2022): *The Münchhausen Method in Type Theory*. In: *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, *LIPIcs* 269, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 10:1–10:20, doi:10.4230/LIPICS.TYPES.2022.10.

[9] Thorsten Altenkirch & Bernhard Reus (1999): *Monadic Presentations of Lambda Terms Using Generalized Inductive Types*. In: *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, *Lecture Notes in Computer Science* 1683, Springer, pp. 453–468, doi:10.1007/3-540-48168-0_32.

[10] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The PoplMark Challenge*. In Joe Hurd & Thomas F. Melham, editors: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, *Lecture Notes in Computer Science* 3603, Springer, pp. 50–65, doi:10.1007/11541868_4.

[11] Nick Benton, Chung-Kil Hur, Andrew Kennedy & Conor McBride (2012): *Strongly Typed Term Representations in Coq*. *J. Autom. Reason.* 49(2), pp. 141–159, doi:10.1007/S10817-011-9219-0.

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

[13] Simon Castellan, Pierre Clairambault & Peter Dybjer (2021): *Categories with Families: Unityped, Simply Typed, and Dependently Typed*, pp. 135–180. Springer International Publishing, doi:10.1007/978-3-030-66545-6_5.

[14] Haskell Brooks Curry & Robert M. Feys (1958): *Combinatory Logic*. 1, North-Holland Publishing Company.

[15] Jason J Hickey (1996): *Formal objects in type theory using very dependent types*. *Foundations of Object Oriented Languages* 3, pp. 117–170. Available at `https://www.cs.cornell.edu/jyh/papers/fool3/paper.pdf`.

[16] Ambrus Kaposi (2023): *Towards quotient inductive-inductive-recursive types*. In: *29th International Conference on Types for Proofs and Programs TYPES 2023–Abstracts*, p. 124. Available at `https://types2023.webs.upv.es/TYPES2023.pdf`.

[17] Chantal Keller & Thorsten Altenkirch (2010): *Hereditary Substitutions for Simple Types, Formalized*. In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, ACM, pp. 3–10, doi:10.1145/1863597.1863601.

[18] Conor McBride (2006): *Type-Preserving Renaming and Substitution*. Available at `http://strictlypositive.org/ren-sub.pdf`.

[19] Hannes Saffrich (2024): *Abstractions for Multi-Sorted Substitutions*. In: *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia*, *LIPIcs* 309, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 32:1–32:19, doi:10.4230/LIPICS.ITP.2024.32.

[20] Hannes Saffrich, Peter Thiemann & Marius Weidner (2024): *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, TyDe 2024, Milan, Italy, 6 September 2024*, ACM, pp. 2–15, doi:10.1145/3678000.3678201.

[21] 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 - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, *Lecture Notes in Computer Science* 9236, Springer, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[22] Kathrin Stark, Steven Schäfer & Jonas Kaiser (2019): *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, CPP 2019, Cascais, Portugal, January 14-15, 2019*, ACM, pp. 166–180, doi:10.1145/3293880.3294101.

[23] The Agda Team (2024): *Agda Documentation*. Available at `https://agda.readthedocs.io/`.

[24] Philip Wadler (2024): *Explicit Weakening*. In: *A Second Soul: Celebrating the Many Languages of Programming - Festschrift in Honor of Peter Thiemann's Sixtieth Birthday, Freiburg, Germany, 30th August 2024*, *EPTCS* 413, pp. 15–26, doi:10.4204/EPTCS.413.2.