The category of simply typed λ-terms in Agda

Chantal Keller
University of Nottingham

3rd July 2008

Abstract. We present a proof in Agda that the substitutions in simply typed λ-calculus form a category which has finite products. We base our syntax on directly typed λ-calculus, without defining first pure λ-calculus. We use parallel substitutions, a better point of view to argue about substitutions. We show that we can reduce all the proofs to similar diagrams, that make it automatic.

Keywords: λ-calculus, substitution, category with finite products, Agda.

1 Introduction

The substitutions in simply-typed λ-calculus [1] form a category which has finite products. This has been proved with different approaches (eg., see [2]). We propose here a new view, with two particularities:

- we use a directly typed syntax, that is to say we consider only typed terms of λ-calculus;
- we prefer parallel substitution to syntactic substitution, because it is a better way to abstract from substitutions and to make simpler proofs.

Parallel substitutions have been introduced by Abadi et al. [3] as a means to manipulate substitutions as abstract objects. This is a better point of view to do proofs on it, than to use the usual substitutions that work on syntactic rules.

The main idea of the proofs we present in this paper is to bring down to proofs for variables as much as possible. Indeed, proofs over variables are far simpler than over terms. We intend to explain how to bring terms down to variables, which is a mechanism that is quite automatic. We notice that this way of doing leads to repetitive proofs that all match with only three diagrams (see section 4.3 for more details).

In section 2, we will introduce the basic syntax of Agda. In section 3, we will present our syntax for λ-calculus and substitutions. Section 4 will be devoted to prove the categorical behaviour of substitutions.

2 A short introduction to Agda

The programs in this paper are terms in a dependent type theory. The development was made in Agda, a dependently typed programming language with good support for programming with inductively defined families of types [4]. It uses a Haskell-like syntax that allows type dependence.

Here is a short introduction to Agda basic syntax. A complete tutorial is available on the agda wiki [5].

2.1 Datatype declarations

Inductive datatypes are defined by a formation rule and constructors. For instance, one could define propositional equality as follows:

```
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a
```

For a given set A, the equality on this set is defined as a set of elements that are identical. The use of the braces permits to define arguments as implicit. The syntax _≡_ means that this relation is infix : we can write a ≡ a.

2.2 Function declarations

We can now define a first function, which will consist in proving that the binary relation defined above is symmetric:

```
sym : {A : Set} → {a b : A} → a ≡ b → b ≡ a
sym refl = refl
```

We first define the type of the function sym, which is a dependent type : given a set A and two elements a and b of A, if we have a proof that a ≡ b, then we can have a proof that b ≡ a. The second line is the definition of the sym function. As A, a and b are implicit arguments, we just have to provide a proof that a ≡ b (in order to obtain a proof that b ≡ a). We do this by pattern matching on this proof. As the only constructor of the equality is refl, we have one single case. The proof then reduces to the constructor refl.

We can prove the relation is transitive the same way:

```
trans : {A : Set} → {a b c : A} → a ≡ b → b ≡ c → a ≡ c
trans refl refl = refl
```

We now have the complete proof that our relation is an equivalence.

3 Explicit substitutions in simply typed λ-calculus

In this section, we present the simply typed λ-calculus with explicit substitutions. We use a typed syntax, which is to say that we will define inductively the typed terms of the λ-calculus, instead of first defining terms and then introducing typing rules. In fact, we are only interested in typed terms.

Substitutions in λ-calculus consist in replacing variables with terms inside a term. We can see terms that are typed in context Δ as trees whose leafs are variables in Δ. Applying a substitution between Γ and Δ to such terms is replacing those leafs with terms that are trees whose leafs are variables in Γ. It is often done by applying some syntactic rules recursively, but here, we prefer parallel substitutions: substitutions are just another means to form terms.

3.1 Syntax

3.1.1 λ-calculus

The set of types Ty : Set is defined with one single base type:

```
data Ty : Set where
  base : Ty
  _=>_ : Ty → Ty → Ty
```

To be able to type variables and terms, we need a set of contexts Context : Set, which are backwards written lists of types:

```
data Context : Set where
  empty : Context
  ext : Context → Ty → Context
```

If Γ is a context and σ a type, the set of the variables that have type σ in context Γ Var Γ σ : Set is defined as follows:

```
data Var : Context → Ty → Set where
  vlast : forall {Γ τ} → Var (ext Γ τ) τ
  weak : forall {σ Γ τ } → Var Γ τ → Var (ext Γ σ) τ
```

And so is the set of the terms that have type σ in context Γ Term Γ σ : Set:

```
data Term : Context → Ty → Set where
  var : forall {Γ σ} → Var Γ σ → Term Γ σ
  lam : forall {Γ τ σ} → Term (ext Γ τ) σ → Term Γ (τ => σ)
  app : forall {Γ τ σ} → Term Γ (τ => σ) → Term Γ τ → Term Γ σ
```

We notice that both Var and Term have the same signature Context → Ty → Set. As a result, variables and terms have lots of similar definitions, and we can abstract from this. If T : Context → Ty → Set, we will be able to define substitutions that replace variables with elements of type T provided three functions exist:

- vr : forall {Γ σ} → Var Γ σ → T Γ σ
- tm : forall {Γ σ} → T Γ σ → Term Γ σ
- wk : forall {Γ σ τ} → T Γ σ → T (ext Γ τ) σ

To compose substitutions, we will need a fourth function:

- subst : forall {Γ Δ σ} → T Δ σ → Subst T Γ Δ → T Γ σ

These functions can form a kit, as suggested in [6], that we will instantiate for variables and for terms. To represent this kit, we use record types:

```
record SubstKit (T : Context → Ty → Set) : Set where
  field
    vr : forall {Γ σ} → Var Γ σ → T Γ σ
    tm : forall {Γ σ} → T Γ σ → Term Γ σ
    wk : forall {Γ σ τ} → T Γ σ → T (ext Γ τ) σ


record SubstKit+ (T : Context → Ty → Set) : Set where
  field
    kit : SubstKit T
    subst : forall {Γ Δ σ} → T Δ σ → Subst T Γ Δ → T Γ σ
```

3.1.2 Substitutions

Given T : Context → Ty → Set and two contexts Γ and Δ, we can define the set of substitutions that transforms elements of T Δ into elements of T Γ as follows:

```
data Subst (T : Context → Ty → Set) : Context → Context → Set where
  substEmpty : forall {Γ} → Subst T Γ empty
  _,_ : forall {Γ Δ σ} → Subst T Γ Δ → T Γ σ → Subst T Γ (ext Δ σ)
```

3.1.3 Categorical combinators

The categorical combinators consist in weakening and lifting substitutions, the identity substitution, and the composition of substitutions.

Weakening a substitution consists in extending the codomain:

```
_+_ : forall {σ T Γ Δ} → Subst T Γ Δ → SubstKit T → Subst T (ext Γ σ) Δ
substEmpty + _ = substEmpty
(u , t) + k = (u + k) , ((SubstKit.wk k) t)
```

Lifting a substitution consists in extending both the domain and the codomain:

```
_++_ : forall {σ T Γ Δ} → Subst T Γ Δ → SubstKit T →
    Subst T (ext Γ σ) (ext Δ σ)
u ++ k = (u + k) , ((SubstKit.vr k) vlast)
```

We can now define the identity substitutions. We first define an identity substitution for variables, then lift it using the constructor var to obtain the identity substitution for terms.

```
idSVar : {Γ : Context} → Subst Var Γ Γ
idSVar {empty} = substEmpty
idSVar {ext Γ σ} = idSVar ++ vk


substTermOfSubstVar : forall {Γ Δ} → Subst Var Γ Δ → Subst Term Γ Δ
substTermOfSubstVar substEmpty = substEmpty
substTermOfSubstVar (s , v) = (substTermOfSubstVar s) , (var v)


idSTerm : {Γ : Context} → Subst Term Γ Γ
idSTerm = substTermOfSubstVar idSVar
```

The composition of two substitutions of the same signature can be defined as follows:

```
compS : forall {T Γ Δ Θ} → SubstKit+ T → Subst T Γ Δ → Subst T Δ Θ →
    Subst T Γ Θ
compS _ _ substEmpty = substEmpty
compS k+ s (s' , t) = (compS k+ s s') , ((SubstKit+.subst k+) t s)
```

3.2 Substitution functions

3.2.1 Variables

If T : Context → Ty → Set, we can apply a Subst T to any variable, and then obtain an element of type T:

```
substVar : forall {T Γ Δ σ} → Subst T Γ Δ → Var Δ σ → T Γ σ
substVar substEmpty ()
substVar (s , t) vlast = t
substVar (s , t) (weak v) = substVar s v
```

```
_[_] : forall {T Γ Δ σ} → Var Δ σ → Subst T Γ Δ → T Γ σ
v [ s ] = substVar s v
```

We have now all the tools to define kits for variables:

```
vk : SubstKit Var
vk = record
  { vr = (\a → a)
  ; tm = var
  ; wk = weak
  }


vk+ : SubstKit+ Var
vk+ = record
  { kit = vk
  ; subst = _[_]
  }
```

3.2.2 Terms

If T : Context → Ty → Set, we can apply a Subst T to any term, and then obtain a term:

```
substTerm : forall {T Γ Δ σ} → SubstKit T → Subst T Γ Δ → Term Δ σ →
    Term Γ σ
substTerm k s (var v) = SubstKit.tm k (v [ s ])
substTerm k s (lam t) = lam (substTerm k (s ++ k) t)
substTerm k s (app t₁ t₂) = app (substTerm k s t₁) (substTerm k s t₂)
```

We will need to weaken terms to complete our kits. Weakening a variable is just applying the constructor weak; but to weaken a term t, we substitute the weakened idSVar to t (this will allow us to bring proofs for terms down to proofs for variables):

```
termWeak : forall {τ Γ σ} → Term Γ σ → Term (ext Γ τ) σ
termWeak t = substTerm vk (idSVar + vk) t
```

We can now define kits and more pleasant notations:

```
tk : SubstKit Term
tk = record
  { vr = var
  ; tm = (\a → a)
  ; wk = termWeak
  }


_[_]₁ : forall {Γ Δ σ} → Term Δ σ → Subst Var Γ Δ → Term Γ σ
t [ u ]₁ = substTerm vk u t


_[_]₂ : forall {Γ Δ σ} → Term Δ σ → Subst Term Γ Δ → Term Γ σ
t [ u ]₂ = substTerm tk u t


tk+ : SubstKit+ Term
tk+ = record
  {kit = tk
  ; subst = _[_]₂
  }
```

We can also have corresponding notations for composition:

```
_∘1_ : forall {Γ Δ Θ} → Subst Var Γ Δ → Subst Var Δ Θ → Subst Var Γ Θ
s ∘1 s' = compS vk+ s s'


_∘2_ : forall {Γ Δ Θ} → Subst Term Γ Δ → Subst Term Δ Θ → Subst Term Γ
    Θ
s ∘2 s' = compS tk+ s s'
```

3.3 Extra functions

We will also need to compose substitutions of different signatures, ie variable and term substitutions, in both senses:

```
_∘3_ : forall {Γ Δ Θ} → Subst Var Γ Δ → Subst Term Δ Θ → Subst Term Γ Θ
_ ∘3 substEmpty = substEmpty
s ∘3 (s' , t) = (s ∘3 s') , (t [ s ]1)


_∘4_ : forall {Γ Δ Θ} → Subst Term Γ Δ → Subst Var Δ Θ → Subst Term Γ Θ
_ ∘4 substEmpty = substEmpty
s ∘4 (s' , v) = (s ∘4 s') , (v [ s ])
```

4 The substitutions form a category which has finite products

In this section, we intend to prove that the structure we defined in section 3 is a category with finite products. We will first prove it for variable substitutions, then for term substitutions, by bringing down to variable cases. But first, we need a few preliminary lemmas.

All the proofs have been checked with Agda. The complete proof is available online [7].

4.1 Preliminary lemmas

Five lemmas are used to prove the compatibility of the structural equality with constructors such as weak, var, lam, app and substExt. They are respectively named reflWeak, reflVar, reflLam, reflApp and reflSubstExt. They are as easy to prove as sym and trans, and have the following prototype:

```
reflCons : forall {p1.1 p1.2 ... pn.1 pn.2} ->
                  p1.1 ≡ p1.2 -> ... -> pn.1 ≡ pn.2 ->
                  Cons p1.1 ... pn.1 ≡ Cons p1.2 ... pn.2
```

We will also need to prove that, if s_1 and s_2 are two structurally equal substitutions, and t_1 and t_2 are two structurally equal terms or variables, then $t_1[s_1] \equiv t_2[s_2]$. This drives to three lemmas that can be proved like the former ones:

```
reflSubst : forall {Δ σ} → {t1 t2 : Term Δ σ} → forall {Γ} →
                   {s1 s2 : Subst Term Γ Δ} → s1 ≡ s2 → t1 ≡ t2 →
                 t1 [ s1 ]2 ≡ t2 [ s2 ]2
```

```
reflSubst2 : forall {Δ σ} → {t1 t2 : Var Δ σ} → forall {T Γ} →
                   {s1 s2 : Subst T Γ Δ} → s1 ≡ s2 → t1 ≡ t2 →
                 t1 [ s1 ] ≡ t2 [ s2 ]
```

```
reflSubst3 : forall {Δ σ} → {t1 t2 : Term Δ σ} → forall {Γ} →
                   {s1 s2 : Subst Var Γ Δ} → s1 ≡ s2 → t1 ≡ t2 →
                 t1 [ s1 ]1 ≡ t2 [ s2 ]1
```

4.2 Proofs for variables

Here we will present the main ideas to prove that substitutions for variables form a category with finite products.

4.2.1 The identity is neutral

We have to show that the identity is neutral for the composition of substitutions, both on the left and on the right. The proofs will be completely different, as the definition of composition is non symmetric on the first and on the second substitutions.

The identity is neutral on the left We first need to prove this property:

Proposition 1 *If v is a variable, s a substitution and σ a type, then:*

$$v[s^{+\sigma}] \equiv (v[s])^{+\sigma}$$

This is proved by simple pattern matching on v:

```
eqWeak : forall {Γ Δ σ τ} → (s : Subst Var Γ Δ) → (v : Var Δ σ) →
                    v [ _+_ {τ} s vk ] ≡ weak (v [ s ])
eqWeak substEmpty ()

eqWeak (s , v₁) vlast = refl
eqWeak (s , v₁) (weak v₂) = eqWeak s v₂
```

We can then mutually prove that:

Proposition 2 *Given v a variable:*

$$\begin{cases} v[id] \equiv v \\ v[id^{+\sigma}] \equiv v^{+\sigma} \end{cases}$$

Once more, pattern matching makes the proof really simple:

```
nLVar : forall {Γ σ} → (v : Var Γ σ) → v [ idSVar ] ≡ v
nLVar vlast = refl
nLVar (weak v) = eqId v
```

```
eqId : forall {Γ σ τ} → (v : Var Γ σ) →           v [ _+_ {τ} idSVar vk ] ≡
     weak v
eqId v = trans (eqWeak idSVar v) (reflWeak (nLVar v))
```

This leads automatically to the main theorem:

Theorem 1 *If s is a variable substitution:*

$$id \circ s \equiv s$$

```
neutralLVar : forall {Γ Δ} → (s : Subst Var Γ Δ) → idSVar ∘₁ s ≡ s
neutralLVar substEmpty = refl
neutralLVar (s , v) = reflSubstExt (neutralLVar s) (nLVar v)
```

The identity is neutral on the right We need one single lemma:

Proposition 3 *Given two substitutions s and s' and a variable v:*

$$(s, v) \circ s'^{+\sigma} \equiv s \circ s'$$

```
nRVar : forall {Γ Δ Θ σ} → (s : Subst Var Γ Δ) → (s' : Subst Var Δ Θ) →
      (v : Var Γ σ) → (s , v) ∘₁ (s' + vk) ≡ s ∘₁ s'
nRVar _ substEmpty _ = refl
nRVar s (s' , _) v = reflSubstExt (nRVar s s' v) refl
```

Then we have our main theorem:

Theorem 2 *If s is a variable substitution:*

$$s \circ id \equiv s$$

```
neutralRVar : forall {Γ Δ} → (s : Subst Var Γ Δ) → s ∘₁ idSVar ≡ s
neutralRVar substEmpty = refl
neutralRVar (s , v) = reflSubstExt (trans (nRVar s idSVar v) (neutralRVar
    s)) refl
```

4.2.2 The composition of substitutions is associative

We just have to previously prove that applying the composition of two substitutions is the same as applying the first one, then the second one:

Proposition 4 *s and s' are two substitutions, v is a variable.*

$$v[s \circ s'] \equiv (v[s'])[s]$$

We prove it by pattern matching on s' and v:

```
aCSVar : forall {Γ Δ Θ σ} → (u : Subst Var Γ Δ) → (v : Subst Var Δ Θ) →
      (v' : Var Θ σ) → v' [ u ∘₁ v ] ≡ (v' [ v ]) [ u ]
aCSVar _ substEmpty ()
aCSVar _ (_ , _) vlast = refl
aCSVar u (v , _) (weak v') = aCSVar u v v'
```

The theorem is now provable:

Theorem 3 *u, v and w are three variable substitutions.*

$$(u \circ v) \circ w \equiv u \circ (v \circ w)$$

```
assoCompSVar : forall {Γ Δ Θ Ξ} → (u : Subst Var Γ Δ) →
                    (v : Subst Var Δ Θ) → (w : Subst Var Θ Ξ) →
                      (u ∘₁ v) ∘₁ w ≡ u ∘₁ (v ∘₁ w)
assoCompSVar _ _ substEmpty = refl
assoCompSVar u v (w , t) = reflSubstExt (assoCompSVar u v w) (aCSVar u v t)
```

4.2.3 This category has finite products

The constructor _,_ has prototype Subst Var Γ Δ −> Var Γ σ −> Subst Var Γ (ext Δ σ). By de-curryfying this function, we obtain a function of prototype Subst Var Γ Δ × Var Γ σ −> Subst Var Γ (ext Δ σ).

We can consider the product Subst Var Γ Δ × Var Γ σ as a product in the category theoretic sense. To do so, we need to find two functions π_1 and π_2 that satisfy the following properties: for all substitution u and variable v,

- $\pi_1(u, v) = u$
- $\pi_2(u, v) = v$
- $(\pi_1(u), \pi_2(u)) = u$

We propose:

- $\pi_1(u) = id^{+\sigma} \circ u$
- $\pi_2(u) = vlast[u]$

We can now prove the required properties:

Theorem 4 $\pi_1(u, v) = u$

This is simply done by using previously proved theorems:

```
π₁ : forall {Γ Δ σ} → (s : Subst Var Γ Δ) → (v : Var Γ σ) →
                (s , v) ∘₁ (idSVar + vk) ≡ s
π₁ s v = trans (nRVar s idSVar v) (neutralRVar s)
```

Theorem 5 $\pi_2(u, v) = v$

This is pure reflexivity:

```
π₂ : forall {Γ Δ σ} → (s : Subst Var Γ Δ) → (v : Var Γ σ) →
                vlast [ s , v ] ≡ v
π₂ _ _ = refl
```

Applying π_2 makes sense only if its argument is an extended substitution. The third theorem then is the following one:

Theorem 6 $(\pi_1(u, v), \pi_2(u, v)) = (u, v)$

Here is the proof in Agda:

```
sp : forall {Γ Δ σ} → (s : Subst Var Γ Δ) → (v : Var Γ σ) →
                ((s , v) ∘₁ (idSVar + vk)) , (vlast [ s , v ]) ≡ s , v
sp s v = reflSubstExt (π₁ s v) (π₂ s v)
```

4.3 Proofs for terms

Here we will present the main ideas to prove that substitutions for terms form a category with finite products. The main idea of all the proofs we will present is to bring down to variable cases, on which proofs are simpler. For instance, to prove that the composition of term substitutions is associative, we first study composition of a term substitution and a variable one.

There are three main ways of conducting proofs:

- by pattern matching on variables: these are similar to the proofs presented in section 4.2;
- by pattern matching on terms. Given a substitution u, a term t, and two expressions f and g that contain u and t, the outline of the proof is the following one:

```
proof : forall {Γ Δ σ} -> (u : Subst Term Γ Δ) -> (t : Term Δ σ) ->
         f u t ≡ g u t

proof substEmpty (var ())
proof (_ , _) (var vlast) = refl
proof (u , _) (var (weak v)) = proof u (var v)

proof u (lam t) = reflLam (trans (proof (u ++ tk) t) ...)

proof u (app t₁ t₂) = reflApp (proof u t₁) (proof u t₂)
```

- by pattern matching on substitutions. Given a substitution u and two expressions f and g that contain u, the outline of the proof is the following one:

```
proof2 : forall {Γ Δ} -> (u : Subst Term Γ Δ) -> f u ≡ g u
proof2 substEmpty = refl
proof2 (u , t) = reflSubstExt (proof2 u) ...
```

The proofs for terms may be really repetitive. In the following sections, we will present the required lemmas and explain which kind of proof described above they refer to. One can check the complete proof to find more details.

4.3.1 The identity is neutral

We have to show that the identity is neutral for the composition of substitutions, both on the left and on the right. Once more, the proofs will be completely different, as the definition of composition is non symmetric on the first and on the second substitutions.

The identity is neutral on the left Defining the identity substitution for terms as a lifting of the identity substitution for variables with the constructor var, we can use proofs for variables to prove that the identity substitution for terms is neutral on the left.

We first have to prove that _+_ and substTermOfSubstVar commute:

```
substWeakExchange : forall {Γ Δ σ} → (s : Subst Var Γ Δ) →
    _+_ {σ} (substTermOfSubstVar s) tk ≡ substTermOfSubstVar (s + vk)
```

This is proved by pattern matching on the substitution and using the lemma eqId.

We also want to prove that _[_] and substTermOfSubstVar commute:

```
substTermAndVar : forall {Γ Δ σ} → (s : Subst Var Γ Δ) → (t : Term Δ σ)
    → t [ substTermOfSubstVar s ]₂ ≡ t [ s ]₁
```

This is done by pattern matching on the term.

As for variables, we have to prove that applying the identity substitution to a term does not change this term:

Proposition 5 *t is a term.*

$$t[id] \equiv t$$

```
nLTerm2 : forall {Γ σ} → (t : Term Γ σ) → t [ idSVar ]₁ ≡ t
nLTerm2 (var v) = reflVar (nLVar v)
nLTerm2 (lam t) = reflLam (nLTerm2 t)
nLTerm2 (app t₁ t₂) = reflApp (nLTerm2 t₁) (nLTerm2 t₂)


nLTerm : forall {Γ σ} → (t : Term Γ σ) → t [ idSTerm ]₂ ≡ t
nLTerm t = trans (substTermAndVar idSVar t) (nLTerm2 t)
```

We can now have our main theorem:

Theorem 7 *u is a term substitution.*

$$id \circ u \equiv u$$

```
neutralLTerm : forall {Γ Δ} → (u : Subst Term Γ Δ) → idSTerm ∘₂ u ≡ u
```

The proof is exactly the same as for variables.

The identity is neutral on the right The proof is exactly the same as for variables. We will only provide the prototypes of the functions.

Theorem 8 *u is a term substitution.*

$$u \circ id \equiv u$$

```
nRTerm : forall {Γ Δ Θ σ} → (u : Subst Term Γ Δ) → (s : Subst Var Δ Θ)
    → (t : Term Γ σ) → (u , t) ∘₂ (substTermOfSubstVar (s ⁺ vk)) ≡
                u ∘₂ (substTermOfSubstVar s)
```

```
neutralRTerm : forall {Γ Δ} → (u : Subst Term Γ Δ) → u ∘₂ idSTerm ≡ u
```

4.3.2 The composition of substitutions is associative

To do this proof, we will bring down to proofs on variables. As a result, for each intermediate result, there will be a series of lemmas, that demonstrate the same result, but once for variables, once for terms, and sometimes mixing terms and variables.

We will need three series of lemmas, that can be deduced from one another. Each lemma deals with substitutions over terms and variables, in such a way that lemmas for terms are deduced from lemmas for variables and lemmas for variables are easily provable.

These three series are:

- aCSTerm: applying the composition of two (variable or term) substitutions to a term is like applying the first one then the second one (really similar to aCSVar):

  Proposition 6 *t is a term or a variable, u and v are substitutions.*

  $$t[u \circ v] \equiv (t[v])[u]$$

- eqWeakTerm, that describes the behaviour of weakening a substituted term (really similar to eqWeak):

  Proposition 7 *Given t a term or a variable, u a substitution and σ a type:*

  $$(t[u])^{+\sigma} \equiv t^{+\sigma}[u^{++\sigma}]$$

- compWeak: lifting the composition of two (variable or term) substitutions is structurally equal to composing the two lifted substitutions:

  Proposition 8 *u and v are two substitutions, σ is a type.*

  $$\begin{cases} (u \circ v)^{+\sigma} \equiv u^{++\sigma} \circ v^{+\sigma} \\ (u \circ v)^{++\sigma} \equiv u^{++\sigma} \circ v^{++\sigma} \end{cases}$$

aCSTerm We first have to prove that the weakened identity just weakens substitutions, for both variables and terms:

Proposition 9 *Given u a (variable or term) substitution, and σ a type:*

$$\begin{cases} id^{+\sigma} \circ u & \equiv & u^{+\sigma} \\ u^{+\sigma} & \equiv & u^{++\sigma} \circ id^{+\sigma} \end{cases}$$

This is proved by pattern matching on the substitution u, and corresponds in the Agda program to the lemmas weakIn.

Transitivity automatically leads to this property:

Proposition 10 *Given u a (variable or term) substitution, and σ a type:*

$$id^{+\sigma} \circ u \equiv u^{++\sigma} \circ id^{+\sigma}$$

We can now prove our main property, it is to say that applying the composition of two substitutions to a term is like applying the first one then the second one, for each kind of substitutions (**Proposition 6**).

This is done by pattern matching on term t and using the series of lemmas called compWeak. Here is one example in Agda:

```
aCSTerm : forall {Γ Δ Θ σ} → (u : Subst Term Γ Δ) →
                             (v : Subst Term Δ Θ) → (t : Term Θ σ) →
                              t [ u ∘₂ v ]₂ ≡ (t [ v ]₂) [ u ]₂
aCSTerm u substEmpty (var ())
aCSTerm u (s , t) (var vlast) = refl
aCSTerm u (s , t) (var (weak v)) = aCSTerm u s (var v)
```

```
aCSTerm u s (lam t) = reflLam
                          (trans (reflSubst {_} {_} {t} {t} (compWeak u s)
                              refl)
                           (aCSTerm (u ++ tk) (s ++ tk) t))
aCSTerm u s (app t₁ t₂) = reflApp (aCSTerm u s t₁) (aCSTerm u s t₂)
```

eqWeakTerm Using symmetry, transitivity and compatibility with substitution, we can obtain this property (**Proposition 7**) thanks to the series of lemmas aCSTerm.

Proof

$$
\begin{array}{rcll}
(t[u])^{+\sigma} & \equiv & (t[u])[id^{+\sigma}] & \text{by definition} \\
 & \equiv & t[id^{+\sigma} \circ u] & \text{by } \textbf{Proposition 6} \\
 & \equiv & t[u^{++\sigma} \circ id^{+\sigma}] & \text{by } \textbf{Proposition 10} \\
 & \equiv & (t[id^{+\sigma}])[u^{++\sigma}] & \text{by } \textbf{Proposition 6} \\
 & \equiv & t^{+\sigma}[u^{++\sigma}] & \text{by definition}
\end{array}
$$

□

compWeak We want to prove that lifting the composition of two (variable or term) substitutions is structurally equal to composing the two lifted substitutions (**Proposition 8**).

The second property automatically comes from the first one (it consists in applying the definition of _++_) and the first property can be proved by pattern matching on the substitution v and using eqWeakTerm. Here is one example in Agda:

```
compWeakAux : forall {Γ Δ Θ σ} → (u : Subst Term Γ Δ) →
          (v : Subst Term Δ Θ) →
          _+_ {σ} (u ∘₂ v) tk ≡ (u ++ tk) ∘₂ (v + tk)
compWeakAux u substEmpty = refl
compWeakAux u (v , t) = reflSubstExt (compWeakAux u v) (eqWeakTerm u t)


compWeak : forall {Γ Δ Θ σ} → (u : Subst Term Γ Δ) →
          (v : Subst Term Δ Θ) →
          _++_ {σ} (u ∘₂ v) tk ≡ (u ++ tk) ∘₂ (v ++ tk)
compWeak u v = reflSubstExt (compWeakAux u v) refl
```

Main theorem Thanks to the lemma aCSTerm applied to term substitutions, we can prove our main theorem exactly the same way we proved it for variables:

Theorem 9

$$(u \circ v) \circ w \equiv u \circ (v \circ w)$$

```
assoCompSTerm : forall {Γ Δ Θ Ξ} → (u : Subst Term Γ Δ) →
          (v : Subst Term Δ Θ) → (w : Subst Term Θ Ξ) →
          (u ∘₂ v) ∘₂ w ≡ u ∘₂ (v ∘₂ w)
assoCompSTerm u v substEmpty = refl
assoCompSTerm u v (w , t) = reflSubstExt (assoCompSTerm u v w) (aCSTerm u v
     t)
```

4.3.3 This category has finite products

We can observe the same properties as for variables concerning finite products. The theorems required to prove that this category has finite products are as simple to prove as for variables, except for the first projector. As he involves the identity substitution for terms, we have to bring it down to variables, as in section 4.3.1.

First projector We will have to first prove that the composition of an extended substitution and the weakened identity is the substitution. We can do this by mutual recursion, demonstrating these two facts:

- the identity substitution for variables is neutral on the right when composing with a term substitution;
- the composition of an extended substitution and a weakened substitution is the composition of the two main substitutions.

The proof relies on pattern matching on substitutions:

```
neutralRTermVar : forall {Γ Δ} → (u : Subst Term Γ Δ) → u ∘4 idSVar ≡ u
neutralRTermVar substEmpty = refl
neutralRTermVar (u , t) = reflSubstExt (extWeakId u t) refl
```

```
extWeak : forall {Γ Δ Θ σ} → (u : Subst Term Γ Δ) →
        (s : Subst Var Δ Θ) → (t : Term Γ σ) →
        (u , t) ∘4 (s + vk) ≡ u ∘4 s
extWeak _ substEmpty _ = refl
extWeak u (s , _) t = reflSubstExt (extWeak u s t) refl


extWeakId : forall {Γ Δ σ} → (u : Subst Term Γ Δ) → (t : Term Γ σ) →
     (u , t) ∘4 (idSVar + vk) ≡ u
extWeakId u t = trans (extWeak u idSVar t) (neutralRTermVar u)
```

```
weakExtTerm : forall {Γ Δ Θ σ} → (u : Subst Term Γ Δ) →
            (v : Subst Term Δ Θ) → (t : Term Γ σ) →
            (u , t) ∘2 (v + tk) ≡ u ∘2 v
weakExtTerm _ substEmpty _ = refl
weakExtTerm u (v , t') t = reflSubstExt (weakExtTerm u v t)
                                        (trans (sym (aCSTerm2 (u , t) (idSVar +
                                            vk) t'))
                                         (reflSubst {_} {_} {t'} {t'} (extWeakId
                                              u t) refl))
```

We can now prove our main theorem:

Theorem 10 $\pi_1(u, v) = u$

```
π'1 : forall {Γ Δ σ} → (u : Subst Term Γ Δ) → (t : Term Γ σ) →
        (u , t) ∘2 (idSTerm + tk) ≡ u
π'1 u t = trans (weakExtTerm u idSTerm t) (neutralRTerm u)
```

Second projector

Theorem 11 $\pi_2(u, v) = v$

This is pure reflexivity:

```
π'₂ : forall {Γ Δ σ} → (u : Subst Term Γ Δ) → (t : Term Γ σ) →
        (var vlast) [ u , t ]₂ ≡ t
π'₂ _ _ = refl
```

Surjective pairing Applying π_2 makes sense only if its argument is an extended substitution. The third theorem then is the following one:

Theorem 12 $(\pi_1(u, v), \pi_2(u, v)) = (u, v)$

Here is the proof in Agda:

```
sp' : forall {Γ Δ σ} → (u : Subst Term Γ Δ) → (t : Term Γ σ) →
        ((u , t) ∘₂ (idSTerm ⁺ tk)) , ((var vlast) [ u , t ]₂) ≡ u , t
sp' u t = reflSubstExt (π'₁ u t) (π'₂ u t)
```

5 Conclusion

We have a proof of the substitutions in simply-typed λ-calculus forming a category which has finite products. This proof has been completely checked using the proof assistant Agda.

This result is not new, but the approach is interesting.

First of all, we used a directly typed syntax. We leave non typable terms aside, focusing only on what is interesting for our proofs. This leads us to simpler proofs, that are structurally well-organized and automatic.

We stress the fact that defining functions for terms from functions for variables increases considerably this automation for proofs. We define the identity substitution for terms and the way to weaken terms using the identity substitution for variables. As a result, a proof for terms reduces easily to a series of proofs melting terms and variables, and finally to proofs for variables.

Instead of implicit substitutions that are defined recursively on the structure of terms, we prefer parallel substitutions. This allows a more abstract point of view than pointwise substitutions, as it can be seen as a means like another to form typed terms in a context.

We believe this work can lead to a kind of an automation for proofs over simply-typed λ-calculus in Agda. We have to justify this assertion by extending it to many other proofs. At least we produced a reliable Agda code that forms a base to conduce similar demonstrations.

References

[1] H. P. Barendregt. The lambda calculus: Its syntax and semantics. 1985.

[2] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. *Computer Science Logic*, 1999.

[3] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.

[4] U. Norell. Towards a practical programming language based on dependent type theory. *PhD thesis, Chalmers Univ. of Tech.*, 2007.

[5] Agda wiki. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php?n=Main.Documentation`.

[6] Conor McBride. Type-preserving renaming and substitution. *Functionnal Pearl*, 2006.

[7] Proofs in agda. `http://perso.ens-lyon.fr/chantal.keller/Documents-etudes/Stage/Parallel-substitution`.