# Autosubst Manual

May 20, 2016

Formalizing syntactic theories with variable binders is not easy. We present Autosubst, a library for the Coq proof assistant to automate this process. Given an inductive definition of syntactic objects in de Bruijn representation augmented with binding annotations, Autosubst synthesizes the parallel substitution operation and automatically proves the basic lemmas about substitutions. Our core contribution is an automation tactic that computes a normal form for Coq expressions containing substitutions. This allows to solve equations between such terms. This makes the usage of substitution lemmas unnecessary. The tactic is based on our current work on a decision procedure for the equational theory of an extension of the sigma-calculus by Abadi et. al. The library is completely written in Coq and uses Ltac to synthesize the substitution operation.

## 1 Tutorial

We start by importing the AUTOSUBST library.

```
Require Import Autosubst.
```

Using de Bruijn Syntax in Coq, the untyped lambda calculus is usually defined as shown Figure 1. Using AUTOSUBST, we can automatically generate the substitution operation. To do so, we annotate the positions of binders in the term type, since de Bruijn indices are interpreted differently if occuring below a binder. The annotated definition

```
Inductive term : Type :=
  | Var (x : nat)
  | App (s t : term)
  | Lam (s : term).
```

Figure 1: Usual term definition with de Bruijn indices

```
Inductive term : Type :=
  | Var (x : var)
  | App (s t : term)
  | Lam (s : {bind term}).
```

Figure 2: Term definition for AUTOSUBST

```
subst_comp s σ τ : s.[σ].[τ] = s.[σ >> τ]
subst_id s : s.[ids] = s
id_subst x σ : (ids x).[σ] = σ x
rename_subst ξ s : rename ξ s = s.[ren ξ]
```

Figure 3: Substitution Lemmas in `SubstLemmas`

is shown in Figure 2. We write `{bind term}` instead of `term` for the argument type of a constructor to indicate that this constructor serves as a binder for the argument. The type `{bind term}` is definitionally equal to `term` and just serves as a tag interpreted while generating the substitution operation. We also need to tag the constructor that builds variables. We do so by specifying the type of its single argument as `var`, which is definitionally equal to `nat`.

Using this definition of term, we can generate the substitution operation `subst` by declaring an instance of the `Subst` type class using our custom tactic `derive`. This is comparable to the usage of deriving-clauses in Haskell. We also need to define instances for the two auxiliary type classes `Ids` and `Rename`, which define the functions `ids` and `rename`. The function `rename` is only needed for technical reasons[1] and is mostly hidden from the interface. The function `ids` is the identity substitution, which is identical to the variable constructor.

```
Instance Ids_term : Ids term. derive. Defined.
Instance Rename_term : Rename term. derive. Defined.
Instance Subst_term : Subst term. derive. Defined.
```

We can now use the pre-defined generic notations to call the just created substitution operation for `term`. Given substitutions $\sigma$ and $\tau$, that is, values of type `var → term`, we can now write `s.[σ]` for the application of $\sigma$ to the term `s` and $\sigma$ `>>` $\tau$ for the composition of $\sigma$ and $\tau$. The notation `s.[σ]` stands for `subst σ s`. The notation $\sigma$ `>>` $\tau$ is equal to $\sigma$ `>>> subst` $\tau$, where `>>>` is function composition, i.e., `(f >>> g) x = g(f(x))`.

Next, we generate the corresponding substitution lemmas by deriving an instance of the `SubstLemmas` type class. It contains the lemmas depiced in Figure 3. The lemma `subst_comp` states that instead of applying two substitutions in sequence, you can apply the composition of the two. This property is essential and surprisingly difficult to show if done manually. The lemma `rename_subst` is needed to eliminate occurrences of the renaming function `rename`. Renaming can be expressed with ordinary substitutions using the function `ren` which lifts a function on variables `var → var` to a substitution `var → term`. It is defined as `ren ξ := ξ >>> ids`.

```
Instance SubstLemmas_term : SubstLemmas term. derive. Qed.
```

[1]The function rename applies a renaming `var → var` to a term. Since it is possible to give a direct structurally recursive definition of `rename`, we use `rename` to give a structurally recursive definition of `subst`. By simplifying `subst` and afterwards unfolding `up`, it is possible to stumble upon an occurrence of `rename`. We try to prevent this by eagerly replacing `rename ξ` with `subst (ren ξ)` in our automation and simplification tactics.

This was all the boilerplate code needed to start using the library. Let us explore the behavior of substitution on some examples. All variables are replaced by the respective value of the substitution. The term `(Var x).[σ]` simplifies to `σ x`. Substitution is extended to application homomorphically. The term `(App s t).[σ]` simplifies to `App s.[σ] t.[σ]`. When going below a binder, the substitution is changed accordingly. The term `(Lam s).[σ]` simplifies to `Lam s.[up σ]`. The substitution `up σ` is equal to `Var 0 .: (σ >> ren (+1))` where `(+1)` is the renaming increasing every variable by one and `.:` is the stream-cons operator. For `a : X` and `f : var → X`, the expression `a .: f` has type `var → X` and satisfies the following equations.

$$\texttt{(a .: f) 0} = \texttt{a}$$
$$\texttt{(a .: f) (S n)} = \texttt{f n}$$

So `up σ` leaves `0` unchanged and for a variable `S x`, it yields `(σ x).[ren(+1)]` to account for the fact that below the binder, the free variables are shifted by 1.

## Substitutivity

Let us start to use the term language. We can define the reduction relation of the untyped lambda calculus as follows.

```
Inductive step : term → term → Prop :=
| Step_Beta s s' t : s' = s.[t .: ids] → step (App (Lam s) t) s'
| Step_App1 s s' t: step s s' → step (App s t) (App s' t)
| Step_App2 s t t': step t t' → step (App s t) (App s t')
| Step_Lam s s' : step s s' → step (Lam s) (Lam s').
```

The most interesting rule is `Step_Beta`, which expresses beta reduction using the stream-cons operator. That is, the term `s.[t .: ids]` is `s` where the index `0` is replaced by `t` and all other indices are reduced by one. Also note that the rule `Step_Beta` contains a superfluous equation to make it applicable in more situations.

Now let us show a property of the reduction relation, the fact that it is closed under substitutions.

```
Lemma step_subst s s' : step s s' → ∀σ, step s.[σ] s'.[σ].
Proof. induction 1; constructor; subst; now asimpl. Qed.
```

The tactic `asimpl` simplifies expressions containing substitutions using a powerful rewriting system. This suffices to make all the subgoals trivial. The equational subgoal

```
s1.[up σ].[t.[σ] .: ids] = s1.[t .: ids].[σ]
```

created by the application of the constructor `Step_Beta` gives a good impression of the power of `asimpl`. Both sides of the equation are simplified to `s1.[t.[σ] .: σ]`.

## Type Preservation

We conclude the tutorial with a proof of type preservation for the simply typed lambda calculus. This example shows how to prove structural properties of a typing relation.

First, we need simple types. We define a base type `Base` and an arrow type `Arr A B` for functions from `A` to `B`.

```
Inductive type :=
| Base
| Arr (A B : type).
```

Then, we can define the typing judgment.

```
Inductive ty (Γ : var → type) : term → type → Prop :=
| Ty_Var x A :     Γ x = A →
                   ty Γ (Var x) A
| Ty_Lam s A B :   ty (A .: Γ) s B →
                   ty Γ (Lam s) (Arr A B)
| Ty_App s t A B : ty Γ s (Arr A B) → ty Γ t A →
                   ty Γ (App s t) B.
```

We use infinite contexts. This allows us to encode contexts as functions of type `var → type`, which coincides with the type of substitutions. Thus we can reuse the operations and tactics of AUTOSUBST for contexts.

Usually, a type preservation proof starts with a weakening lemma for the typing relation, which states that you can add a binding to the context. In de Bruijn formalizations, it is usually stated with an operation that adds a single binding at an arbitrary position in the context. Using parallel substitutions, we can generalize this to all contexts that can be obtained by reinterpreting the indices. This avoids ugly shiftings in the lemma statement. Moreover, this single lemma subsumes weakening, contraction and exchange.

```
Lemma ty_ren Γ s A: ty Γ s A  → ∀Δ ξ,
                    Γ = (ξ >>> Δ)  →
                    ty Δ s.[ren ξ] A.
Proof.
  induction 1; intros; subst; asimpl; econstructor; eauto.
  - eapply IHty. autosubst.
Qed.
```

For case of typing a lambda expression, the application of `autosubst` solves the following equation between contexts.

```
A .: ξ >>> Δ = (0 .: ξ >>> (+1)) >>> A .: Δ
```

This also happens to be a good example for the somewhat complex but efficient precedence of `.:` and the composition operators. Altough both have the same[2] precedence

[2]Technically, this is not directly possible with the Coq notation mechanism. However, you can achieve the same effect by giving `.:` a lower precedence level (that is, higher precedence) and its right argument the same level as the composition operators. It would be simpler to give everything right associativity, but this does not work for heterogeneous substitutions.

level, the composition operators are left-associative while `.:` is right associative. So the given equation is equivalent to the following.

```
A .: (ξ >>> Δ) = (0 .: (ξ >>> (+1))) >>> (A .: Δ)
```

Unfortunately, Coq 8.4 contains a bug such that the right-hand side is printed without parentheses, although this would be parsed as the ill-typed term

```
0 .: ((ξ >>> (+1)) >>> (A .: Δ)).
```

By generalizing `ty_ren` to substitutions, we obtain that we preserve typing if we replace variables by terms of the same type.

```
Lemma ty_subst Γ s A: ty Γ s A  → ∀σ Δ,
                      (∀x, ty Δ (σ x) (Γ x))  →
                      ty Δ s.[σ] A.
Proof.
  induction 1; intros; subst; asimpl; eauto using ty.
  - econstructor. eapply IHty.
    intros [|]; asimpl; eauto using ty, ty_ren.
Qed.
```

Again, the only non-trivial subgoal is the typing of a lambda expression. Applying the inductive hypothesis yields the following subgoal.

```
∀x : var, ty (scons A Δ) (up σ x) (scons A Γ x)
```

We solve it by destructing `x` with `intros [|]` and simplifying the resulting terms with `asimpl`, which makes them match `Ty_Var` and `ty_ren`, respectively.

To show type preservation of the simply typed lambda calculus, we use `ty_subst` to justify the typing of the result of the beta reduction. The tactic `ainv` performs `inversion` on all hypothesis where this does not produce more than one subgoal.

```
Lemma ty_pres Γ s A : ty Γ s A  → ∀s',
                      step s s'  →
                      ty Γ s' A.
Proof.
  induction 1; intros s' H_step; asimpl;
  inversion H_step; ainv; eauto using ty.
  - eapply ty_subst; try eassumption.
    intros [|]; simpl; eauto using ty.
Qed.
```

Again, we need to destruct the universally quantified variable in the premise of `ty_subst`.

This tutorial only covered the basic aspects of Autosubst. For examples of how to use Autosubst for many-sorted syntax with heterogeneous substitutions or with dependent contexts, please refer to the case studies distributed with Autosubst.

```
Inductive type : Type :=
| TyVar (x : var)
| Arr   (A B : type)
| All   (A : {bind type}).

Inductive term :=
| TeVar (x : var)
| Abs   (A : type) (s : {bind term})
| App   (s t : term)
| TAbs  (s : {bind type in term})
| TApp  (s : term) (A : type).
```

Figure 4: Declaration of the syntax of System F

# 2 Reference Manual

## 2.1 Defining the Syntax

To start using Autosubst, you first have to define an inductive type of terms with de Bruijn indices. This should be a simple inductive definition without dependent types. There must be at most one constructor for variables, aka de Bruijn indices. It must have a single argument of type `var`, which is a type synonym for `nat`. If a constructor acts as a binder for a variable of the term type `T` in a constructor argument of type `U`, then `U` has to be replaced by `{bind T in U}`. We can write `{bind T}` instead of `{bind T in T}`. Figure 4 shows how this looks for the two-sorted syntax of System F.

## 2.2 Generating the Operations

We need to generate the substitution operations for the used term types and the corresponding lemmas. This is done with instance declarations for the corresponding typeclass instances and the tactic `derive`, which is defined as `trivial with derive` and we have collected a tactic for every typeclass in the hint database `derive`. The operations are summarized in Table 1 and the corresponding lemmas in Table 2.

For example, the syntax of System F needs the declarations shown in Figure 5. It is important to build the instances in the right order because they depend on each other. We summarize the dependencies between the type class instances in Table 3.

## 2.3 Defined Operations

Autosubst defines a number of operations, some of which depend on the generated operations. They are important not only because they are useful in statements, but more importantly because our custom tactics incorporate facts about them. They are summarized in Table 4.

| Typeclass | Function | Notation | Type |
|---|---|---|---|
| `Ids term` | `ids x` | | `var → term` |
| `Rename term` | `rename ξ s` | | `(var → var) → term → term` |
| `Subst term` | `subst σ s` | `s.[σ]` | `(var → term) → term → term` |
| `HSubst term1 term2` | `hsubst σ s` | `s.|[σ]` | `(var → term1) → term2 → term2` |

Table 1: Operations that can be generated with AUTOSUBST

| Typeclass | Contained Lemmas |
|---|---|
| `SubstLemmas term` | `rename ξ s = s.[ren ξ], s.[ids] = s, (ids x).[σ] = σ x, s.[σ].[τ] = s.[σ >> τ]` |
| `HSubstLemmas term1 term2` | `s.|[ids] = s, (ids x).|[σ] = ids x, s.|[σ].|[τ] = s.|[σ >> τ]` |
| `SubstHSubstComp term1 term2` | `s.[σ].|[τ] = s.|[τ].[σ >>| τ]` |

Table 2: Lemmas that can be generated with AUTOSUBST

```
Instance Ids_type : Ids type. derive. Defined.
Instance Rename_type : Rename type. derive. Defined.
Instance Subst_type : Subst type. derive. Defined.

Instance SubstLemmas_type : SubstLemmas type. derive. Qed.

Instance HSubst_term : HSubst type term. derive. Defined.

Instance Ids_term : Ids term. derive. Defined.
Instance Rename_term : Rename term. derive. Defined.
Instance Subst_term : Subst term. derive. Defined.

Instance HSubstLemmas_term : HSubstLemmas type term. derive. Qed.
Instance SubstHSubstComp_type_term : SubstHSubstComp type term. derive. Qed.

Instance SubstLemmas_term : SubstLemmas term. derive. Qed.
```

Figure 5: Declarations to derive the operations and lemmas for System F

| Typeclass | Required Prior Declarations |
|---|---|
| `Ids term` | none |
| `Rename term` | none |
| `Subst term` | `Rename term`,<br>`HSubst term' term`<br>if `term` contains `{bind term' in term}` |
| `HSubst term1 term2` | `Subst term1`,<br>`HSubst term3 term4`<br>if `term2` contains `{bind term3 in term4}`,<br>`HSubst term1 term3`<br>if `term2` contains `term3` |
| `SubstLemmas term` | `Ids term`,<br>`Subst term`,<br>`HSubstLemmas term1 term2`<br>and `SubstHSubstComp term1 term2`<br>if `Subst term` requires `HSubst term1 term2` |
| `HSubstLemmas term1 term2` | `HSubst term1 term2`,<br>`SubstLemmas term1` |

Table 3: Required Declaration Order of the Typeclass Instances

| Name | Notation | Definition | Type |
|---|---|---|---|
| funcomp | f >>> g | fun x ⇒ g(f x) | ∀A B C : Type, (A → B) → (B → C) → A → C |
| scons | a .: f | fun x ⇒ match x with \| 0 ⇒ a \| S x' ⇒ f x' end | ∀X : Type, X → (var → X) → var → X |
| scomp | σ >> τ | σ >>> subst τ | (var → term) → (var → term) → var → term |
| hcomp | σ >>\| θ | σ >>> hsubst θ | (var → term1) → (var → term2) → var → term1 |
| ren | ren ξ | ξ >>> ids | (var → var) → var → term |
| lift | (+ n) | fun x ⇒ n + x | var → var → var |
| up | up σ | ids 0 .: σ >> ren(+1) | (var → term) → var → term |

Table 4: Defined Primitives of Autosubst

### 2.4 The Automation Tactics

Autosubst defines two automation tactics: `asimpl` and `autosubst`.

`asimpl`

Normalizes the claim.

`asimpl in H`

Normalizes the hypothesis `H`.

`asimpl in *`

Normalizes the claim and all hypothesis.

`autosubst`

Normalizes the claim and tries to solve the resulting equation.

Both of them normalize the goal using a convergent rewriting system. But while the interface and behavior of `asimpl` mimics `simpl`, the closing tactic `autosubst` first normalizes an equational claim and then tries to prove it. The rewriting system is an extension of the $\sigma$-calculus by Abadi et. al. [1]. Our goal is to solve all equations that hold without assumptions and are built using only our primitives, application and variables. At the moment, we hope to achieve this if `(+n)` is only used with a constant `n`. We consider ever real-world example of an unsolvable such equation a bug and invite you to submit it.

The normalization is done by interleaving the rewriting system with calls to `simpl` to incorporate the definitions of the derived operations.

## 3 Internals

In the following, we describe technical challenges and how we solved them in Coq.

### 3.1 Normalizing Substitutions

To simplify terms containing substitutions, we use a rewriting system based on the convergent $\sigma$-calculus by Abadi et. al. [1]. We extended it to variables for renamings, heterogeneous substitutions and to lifts `(+n)` that add an arbitrary natural number `n` instead of just 1. To keep the rewriting system small, we base it on function composition and a stream-cons that works on arbitrary streams. So first, we replace

- $\sigma$ `>>` $\tau$ with $\sigma$ `>>> subst` $\tau$
- $\sigma$ `>>|` $\tau$ with $\sigma$ `>>> hsubst` $\tau$
- `ren` $\xi$ with $\xi$ `>>> ids`
- `up` $\sigma$ with `ids 0 .:` $\sigma$ `>>> subst ((+1) >>> ids)`.

and will undo these unfoldings in the end. Also, we make function composition right associative, which we also undo in the end. These tricks allow us to reason about `(+n)`, `.:` and `>>>` separately from the proper substitution operations.

## 3.2 Reducible Recursive Type Class Instances

We need the substitution operation to reduce and simplify because there is no other way how our automation tactics could learn about the behavior of substitution on custom term types. However, this is challenging since the substitution operations we derive are instances of a type class. We needed a number of tricks to make this work smoothly.

- We use singleton type classes. So a type class instance is just a definition of the recursive procedure and the type class function reduces to its instance argument. This is important for two reasons. First, the guardedness checker does not unfold the record projections used for non-singleton type classes. Second, when `simpl` reduces a definition bound to a `fix`, it replaces all occurrences of this fix with the name of the definition afterwards. This also just works for singleton type classes.

- All recursive calls are formulated using the function of the type class with the procedure name bound in the `fix`-term serving as the implicit instance argument. This way, the result of the reduction of a type class function contains again calls to this type class function.

- The Coq unification algorithm can perform unfoldings that are impossible with `simpl`. This can lead to implicit instance arguments being unfolded. In turn, the type class inference can no longer infer instances depending on the unfolded instance. Apart from using `simpl` before using tactics that trigger unification like `apply` or `constructor`, the only way to circumvent this is to revert the unfolding of instances. We automatically do this in all automation tactics by reinferring implicit instance arguments using `exact _`.

## 3.3 Generating the Operations Using Ltac

We generate the renaming and substitution operations using Ltac. Since these are recursive functions, we use the tactics `fix` and `destruct`. Consider the substitution operation for the term language from the tutorial. Our `derive` tactic constructs the following (proof) term.

```
fix inst (σ : var → term) (s : term) {struct s} : term :=
  match s as t return (annot term t) with
  | Var x ⇒ σ x
  | App s1 s2 ⇒ App s1.[σ] s2.[σ]
  | Lam s0 ⇒ Lam s0.[up σ]
  end
```

Apart from the return annotation, which is an artifact of our approach, this looks quite clean. However, the recursive call is hidden in the implicit instance argument to `subst`. We can see it if we show all implicit arguments.

```
fix inst (σ : var → term) (s : term) {struct s} : term :=
  match s as t return (@annot Type term term t) with
  | Var x ⇒ σ x
  | App s1 s2 ⇒ App (@subst term inst σ s1) (@subst term inst σ s2)
  | Lam s0 ⇒ Lam (@subst term inst (@up term Ids_term Rename_term σ) s0)
  end
```

To construct this term with Ltac, we start with `fix inst 2` to generate the `fix`-term. Since we want to use the recursive identifier `ident` as a typeclass instance, we make it accessible to the instance inference by changing its type with

```
change _ with (Subst term) in inst
```

Next, we need to construct the `match`, which we can do with a `destruct`. But then, the subgoals do not tell us the constructor of the current `match` case. We get this information by annotating the goal with `s` before calling `destruct`. Then this annotation contains the current constructor with all its arguments after the `destruct`. The function `annot`, which is the identity on the first argument and ignores the second, allows us to perform the annotation. So we use the script `intros ξ s; change (annot term s); destruct s` and then the claims of the subgoals are

- `annot term (Var x)`
- `annot term (App s1 s2)`
- `annot term (Lam s0)`

So in effect, we have gained access to the patterns of the `match`. Using a recursive, value-producing tactic, we can fold over the applied constructor like a list and change every argument depending on its type. The types of the arguments happen to contain the binding annotations in the definition of `term`, so we can use an Ltac-`match` to read them and apply substitutions to the arguments accordingly. The type class inference automatically inserts the recursive call and the guardedness checker is able to unfold `subst` to see the applied recursive call.

# 4 Best Practices

## 4.1 Extending the Automation

If you want to extend the automation to support equations for a new function, you should do the following.

First, try to define the new function using function composition or other existing supported functions. If this is possible, then you should define it using the notation mechanism to prevent the supported functions from being hidden behind a defined name.

Otherwise, you have to extend the built-in tactics `autosubst_unfold` and `fold_comp` to perform the unfolding and folding respectively.

For example, if you want to lift a semantic interpretation to substitutions

`subst_interp : (var → value) → (var → term) → var → value`

then you should define

`Notation subst_interp` $\rho$ $\sigma$ `:=` $\sigma$ `>>> interp` $\rho$`.`

This automatically adds some limited support. To get full support, you can add the required lemmas to the autorewrite database `autosubst`, which is used by the tactics `asimpl` and `autosubst`.

## References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.