Autosubst: Reasoning with de Bruijn Terms and Parallel Substitution

Steven Schäfer Tobias Tebbi Gert Smolka

Saarland University

June 10, 2015

To appear in Proc. of ITP 2015, Nanjing, China, Springer LNAI

Reasoning about syntax with binders plays an essential role in the formalization of the metatheory of programming languages. While the intricacies of binders can be ignored in paper proofs, formalizations involving binders tend to be heavyweight. We present a discipline for syntax with binders based on de Bruijn terms and parallel substitutions, with a decision procedure covering all assumption-free equational substitution lemmas. The approach is implemented in the Coq library AUTOSUBST, which additionally derives substitution operations and proofs of substitution lemmas for custom term types. We demonstrate the effectiveness of the approach with several case studies, including part A of the POPLmark challenge.

1 Introduction

Proofs in the metatheory of programming languages and type systems are the kind of proofs that mathematicians hate for good reason: they are long, contain few essential insights, and have a lot of tedious but error-prone cases. Thus they appear to be an ideal target for computer-verification. Unfortunately, such efforts have often been hampered by the formalization of binders. On paper, it is consensus that issues of α-equivalence can be neglected with a clean conscience. For a machine-verifiable formalization, this is not an option.

We propose a style of reasoning about binders based on parallel de Bruijn substitutions [6]. A (parallel) substitution is a function mapping all variables to terms. The instantiation of a term s under a substitution σ implements capture-avoiding substitution, replacing all free variables simultaneously.

Parallel substitutions by themselves are already a useful tool. De Bruijn [6] used parallel substitutions to produce a simplified proof of the Church-Rosser theorem.

Statements about typing judgements, such as weakening and substitutivity, can be generalized to manipulate the whole context at the same time [9, 2]. When working with logical relations [8], we need parallel substitutions in the statement of the fundamental theorem. In all of these examples it is beneficial to quantify over parallel substitutions.

Parallel de Bruijn substitutions can be described in the σ-calculus of Abadi et. al. [1], which is a first-order equational theory capturing the interaction of terms and substitutions. The σ-calculus is formulated as a confluent and terminating rewriting system. In addition, we recently showed that computing normal forms with the σ-calculus yields a decision procedure for equational substitution lemmas [14].

While none of the ingredients we use are novel, their combination yields a powerful and practical approach to syntactic theories. In particular, the application of the σ-calculus to prove substitution lemmas is novel.

There are a number of routine proofs required to adapt our approach to a given term language. We have implemented a Coq library Autosubst [15] to automate this step. Autosubst can automatically generate the substitution operations for a custom inductive type of terms and prove the corresponding substitution lemmas. Autosubst offers tactics that implement the aforementioned normalization and decision procedure. Autosubst also supports heterogeneous substitutions between multiple syntactic sorts, like terms with type binders in System F.

The paper is structured as follows. In section 2, we present the approach behind Autosubst using the untyped λ-calculus as a running example. We explain our handling of heterogeneous substitutions in section 3. In section 4, we report on our experiences using Autosubst in practical formalizations.

1.1 Evaluation

We evaluate our approach, as well as Autosubst, on a number of case studies.

- *Type preservation of CC_ω*, a Martin-Löf style dependent type theory [11] with a predicative hierarchy of universes. This is a technically challenging development, which needs a number of auxiliary results (e.g., confluence of reduction). In total, the development takes 214 lines of specification and 233 lines of proof.
- *Normalization of System F.* The proof uses logical relations, which require some form of parallel substitutions in the fundamental theorem. We have mechanized both a proof for a call-by-value variant of System F, as well as a proof of strong normalization following Girard [8]. For the call-by-value case we require 99 lines of specification and 65 lines of proof. The strong normalization proof amounts to 153 lines of specification and 107 lines of proof.
- *Progress and type preservation of System F with subtyping.* This is part A of

the POPLmark challenge [4]. The whole development consists of 206 lines of specification and 240 lines of proof. This is less than half the length of existing solutions in Coq. The development has been kept fairly close to the paper proofs for easier comparison to other implementations. If we deviate from the paper proofs and omit well-formedness assumptions, we can shorten the proofs to 185 lines of specification and 157 lines of proof.

We report on these case studies in section 4.

All case studies, as well as the source code of AUTOSUBST can be found at `https://www.ps.uni-saarland.de/autosubst`.

1.2 Related Work

There are a number of libraries and tools supporting proofs about syntax. We mention the ones that are available for Coq. What sets AUTOSUBST apart from the related work is our usage of parallel substitutions, which subsume single variable substitutions. This allows us to offer an automation tactic for solving substitution lemmas and simplifying terms involving substitutions based on a clearly defined equational theory. Additionally, AUTOSUBST is completely implemented within Coq and does not require external tools.

- CFGV [3] uses a generic type of context free grammars with variable binding. Custom term types are obtained by instantiating the generic construction. The library provides a number of general lemmas to work with CFGVs. The representation is named and explicitly deals with issues of α-equivalence.
- DBGen [12] is an external tool that generates de Bruijn substitution operations and proofs of corresponding lemmas. It includes support for syntax defined by mutual recursion. It generates a single-variable substitution operation.
- DBLib [13] is a Coq library for de Bruijn substitutions. It offers a generic type class interface and support for defining single-variable substitution operations in a uniform way. The substitution lemmas are solved by generic tactics. It offers tactics that nicely unfold the substitution operations. Also, DBLib offers some automation in the form of a hint database with frequently needed lemmas.
- GMeta [10] uses a generic term type and automatically generated isomorphisms between it and custom term types. It supports a de Bruijn and a locally nameless interface. Moreover, it has support for mutually recursive syntax and the corresponding heterogeneous substitutions. It supports single-variable substitutions, but lacks automation for substitution lemmas.
- Lambda Tamer [7] is a small library of general purpose automation tactics. It provides support for working with higher-order and dependently-typed abstract syntax.

- LNGen [5] is a generator for single-variable locally-nameless substitution operations and the corresponding lemmas for Coq. It is based on the specification syntax of the Ott tool [16].

2 From de Bruijn to Autosubst

In this section, we illustrate the de Bruijn representation as used in AUTOSUBST on the example of the untyped λ-calculus. We first recall the definition of de Bruijn terms and instantiation (2.1) and argue that this yields a model of the σ-calculus [14]. We then demonstrate reasoning with de Bruijn terms at the example of the Takahashi, Tait, Martin-Löf proof of the Church-Rosser theorem (2.2). In 2.3 we discuss the implementation of de Bruijn terms and substitutions in the interactive theorem prover Coq. This is automated in AUTOSUBST, and in 2.4 we present the same implementation using AUTOSUBST.

2.1 De Bruijn Representation and Substitution

In this section, we recall the de Bruijn representation of terms in the untyped λ-calculus and derive the corresponding definitions and lemmas.

The untyped λ-calculus is the archetype of a language with local definitions. A term of the untyped λ-calculus is either an application, a binder, or a variable. The idea behind a variable is that it is a reference — either to a binder within the same term or to an outside context. In de Bruijn representation we implement these references using natural numbers. The number n refers to the n-th enclosing binder, counting from 0.

We call the numbers by which we implement variables **de Bruijn indices**, or simply **indices**, and write them as x, y, z. We write **terms** $s, t \in \mathbb{T}$ as $s, t ::= x \mid s\,t \mid \lambda.\, s$. A **substitution** is a total function mapping indices to terms. A **renaming** is a substitution that replaces indices by indices. Note that we do not assume that renamings are bijective. The letters σ, τ, θ will denote substitutions, while ξ, ζ will stand for renamings.

A substitution can be seen as an infinite sequence $(s_0, s_1, s_2, \ldots)$ of terms. This view motivates the definition of a cons operation $s \cdot \sigma$.

$$(s \cdot \sigma)(x) := \begin{cases} s & \text{if } x = 0 \\ \sigma(x-1) & \text{otherwise} \end{cases}$$

We also introduce notation for the **identity** and the **shift** substitution:

$$\mathsf{id}(x) := x \qquad\qquad \uparrow(x) := x + 1$$

Next, we define the **instantiation** and **composition** operations $s[\sigma]$ (read s under σ) and $\sigma \circ \tau$, which implement capture-avoiding substitution and composition of substitutions, respectively.

$$\begin{aligned} x[\sigma] &= \sigma(x) \\ (s\,t)[\sigma] &= (s[\sigma])\,(t[\sigma]) \qquad (\sigma \circ \tau)(x) = \sigma(x)[\tau] \\ (\lambda.\, s)[\sigma] &= \lambda.\, (s[\Uparrow\sigma]) \end{aligned}$$

In the third equation, we change the substitution using the operator $\Uparrow$ (pronounced **"up"**). This is necessary because λ is a binder and changes the interpretation of the indices in its scope. As we want to implement capture-avoiding substitution, we have to preserve the bound index 0 and increase all other indices to skip the additional binder. Combining both cases, we define $\Uparrow\sigma$ as

$$\Uparrow\sigma \;:=\; 0 \cdot (\sigma \circ \uparrow)$$

This shows that the operation $\Uparrow$ does not need to be primitive.

This definition of instantiation is by mutual recursion between instantiation, $\Uparrow$ and composition. We have to argue termination to show that it is well-defined.

The mutual recursion can be broken up by considering the definition for renamings. For a renaming ξ, the definition of $\Uparrow\xi$ simplifies to a non-recursive one (for a renaming ξ and a substitution σ, $\xi \circ \sigma$ is ordinary function composition) and thus the definition of $s[\xi]$ boils down to a structural recursion. Since $\uparrow$ is a renaming, we can now define $\Uparrow\sigma$ for general substitutions. Finally, we can define $s[\sigma]$ with a second structural recursion. Such a two-level definition of instantiation has been used by Adams in [2].

De Bruijn terms and parallel substitutions, together with the substitution operations (instantiation, composition, cons, shift, and id) form a model of the σ-calculus by Abadi et. al. [1]. The σ-calculus is a calculus of explicit substitutions. Explicit substitutions were intended to analyze reduction and its implementation in a more fine-grained way. For us, it is interesting for two reasons.

First, the σ-calculus can express all substitutions necessary to describe reductions in the λ-calculus. For instance, β-reduction can be expressed using cons. In de Bruijn representation, a term $(\lambda.\, s)\,t$ reduces to the term s where the index 0 is replaced by t and every other index is decremented by 1, since we removed a binder. Using the substitution operations of the σ-calculus, this is expressed as $(\lambda.\, s)\,t \rhd s[t \cdot \mathsf{id}]$. We can express η-reduction as $(\lambda.\, s[\uparrow]\,0) \rhd s$. The bound index 0 cannot appear in the image of s under the shift substitution, which replaces an explicit side condition on the rule.

Second, the σ-calculus yields a useful decision procedure for equational substitution lemmas. The decision procedure is based on the rewriting system shown in

$$
\begin{aligned}
(s\,t)[\sigma] &\equiv (s[\sigma])(t[\sigma]) & \mathsf{id} \circ \sigma &\equiv \sigma \\
(\lambda.\, s)[\sigma] &\equiv \lambda.\, (s[0 \cdot \sigma \circ \uparrow]) & \sigma \circ \mathsf{id} &\equiv \sigma \\
0[s \cdot \sigma] &\equiv s & (\sigma \circ \tau) \circ \theta &\equiv \sigma \circ (\tau \circ \theta) \\
\uparrow \circ\, (s \cdot \sigma) &\equiv \sigma & (s \cdot \sigma) \circ \tau &\equiv s[\tau] \cdot \sigma \circ \tau \\
s[\mathsf{id}] &\equiv s & s[\sigma][\tau] &\equiv s[\sigma \circ \tau] \\
0[\sigma] \cdot (\uparrow \circ\, \sigma) &\equiv \sigma & 0 \cdot \uparrow &\equiv \mathsf{id}
\end{aligned}
$$

Figure 1: The convergent rewriting system of the σ-calculus

Figure 1. As shown in [14], it is also complete for the given model. Thus, we obtain a rewriting-based decision procedure for equations containing the operations defined in this section, the term constructors of the untyped λ-calculus, and universally quantified meta-variables for terms and substitutions.

This concludes the definitions needed for the untyped λ-calculus. It is interesting to note that the use of parallel substitutions goes back to de Bruijn [6]. Nevertheless, the use of single-index substitutions is widespread in the programming language community, although they are equally complicated to define, significantly less expressive, and their definition involves ad-hoc recursive "shift" functions. We believe that single-index substitutions are responsible for much of the dissatisfaction with de Bruijn terms. In contrast, the careful choice of substitution operations by Abadi et. al. [1] makes this approach to formalizing syntax elegant.

2.2 Case Study: Confluence of Reduction

De Bruijn terms were originally introduced both as an implementation technique and as a way of simplifying paper proofs about the λ-calculus [6]. In this section we will outline a formal proof of the Church-Rosser theorem based on de Bruijn terms. There is almost no overhead in the definitions when compared to a presentation with named variables. We illustrate how parallel substitutions are used to obtain useful generalizations and how substitution lemmas can be shown using the σ-calculus.

The Church-Rosser theorem is equivalent to the statement that reduction is **confluent**, that is, if a term s reduces to t_1 and to t_2 in an arbitrary number of steps, then we can always find a common reduct of t_1 and t_2. The following proof is based on the work of Takahashi, Tait, and Martin-Löf [17, 11].

Definition 1 **Parallel reduction** $s \rhd t$ is defined by the following system of inference rules.

$$\frac{s_1 \rhd s_2 \qquad t_1 \rhd t_2}{(\lambda.\, s_1)\, t_1 \rhd s_2[t_2 \cdot \mathsf{id}]} \qquad \frac{}{x \rhd x} \qquad \frac{s_1 \rhd s_2 \qquad t_1 \rhd t_2}{s_1\, t_1 \rhd s_2\, t_2} \qquad \frac{s_1 \rhd s_2}{\lambda.\, s_1 \rhd \lambda.\, s_2}$$

Parallel reduction on substitutions is defined pointwise.

$$\sigma \rhd \tau := \forall x,\, \sigma(x) \rhd \tau(x)$$

Parallel reduction interpolates between ordinary reduction and many-step reduction, that is, $\triangleright \subseteq \rhd \subseteq \triangleright^*$. From this, we see that the confluence of parallel reduction implies the confluence of single-step reduction.

Parallel reduction may contract an arbitrary number of toplevel redexes in a single step. We consider a function ρ which performs a **maximal parallel reduction**.

$$\begin{aligned} \rho(x) &= x \\ \rho(\lambda.\, s) &= \lambda.\, \rho(s) \\ \rho((\lambda.\, s)\, t) &= \rho(s)[\rho(t) \cdot \mathsf{id}] \\ \rho(s\, t) &= \rho(s)\, \rho(t) && \text{if } s \text{ is not a } \lambda \text{ abstraction} \end{aligned}$$

The result of applying ρ to a term s is maximal in the sense that we can always extend an arbitrary parallel reduction from s to ρs by contracting the remaining redices.

Lemma 2 (Triangle property) For terms s, t, if $s \rhd t$, then $t \rhd \rho(s)$.

From this, the confluence of parallel reduction follows by a diagram chasing argument. The proof of Lemma 2 relies on a strong substitutivity property for parallel reductions. We need to show that if $s_1 \rhd s_2$ and $t_1 \rhd t_2$ then $s_1[t_1 \cdot \mathsf{id}] \rhd s_2[t_2 \cdot \mathsf{id}]$.

We cannot show this lemma by a direct induction, because the β-substitution $t_1 \cdot \mathsf{id}$ is not stable when going under a binder. In the case of a $\lambda.\, s_1$, we would have to show by induction that $s_1[\Uparrow(t_1 \cdot \mathsf{id})] \rhd s_2[\Uparrow(t_2 \cdot \mathsf{id})]$. The minimal generalization that works is to consider all **single index substitutions**, that is, substitutions which replace an index k by a term s. In our notation, these substitutions can be written as $\Uparrow^k(s \cdot \mathsf{id})$. If we continue in this vein, we will notice that we also have to show a similar lemma for shift substitutions of the form $\Uparrow^k(\uparrow^l)$. A better strategy is to generalize the lemma over all substitutions.

Lemma 3 (Strong Substitutivity) For all terms $s \rhd t$ and substitutions $\sigma \rhd \tau$ we have $s[\sigma] \rhd t[\tau]$.

The proof proceeds by induction on the derivation of $s \rhd t$. In the case of the binder, we need to show that $\sigma \rhd \tau$ implies $\Uparrow\sigma \rhd \Uparrow\tau$. This statement in turn depends on a special case of the same lemma. In particular, we can show it by Lemma 3 specialized to renamings. Intuitively, this is because the proof has to follow the inductive structure of the instantiation operation.

The other interesting case in the proof of Lemma 3 is the case for β-reduction, where we have to show a substitution lemma.

$$s[\Uparrow\tau][t[\tau] \cdot \mathsf{id}] = s[t \cdot \mathsf{id}][\tau]$$

This equation holds as a consequence of the axioms of the σ-calculus. In particular we can show it completely mechanically by rewriting both sides of the equation to the normal form $s[t[\tau] \cdot \tau]$.

2.3 Realization in Coq

In Coq we can define the terms of the untyped λ-calculus as an inductive type.

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

The cons operation can be defined generically for functions $\mathbb{N} \to X$ for every type X. We introduce the notation `s .: σ` to stand for $s \cdot \sigma$. The identity substitution id will be written as `ids` and corresponds to the variable constructor. By post-composing with the identity substitution, we can lift arbitrary renamings (functions `ξ : nat → nat`) to substitutions. Since there is no standard notation for forward composition, we introduce the notation `f >>> g` for forward composition of functions. For readability, we also introduce notation for the coercion of renamings into substitutions, `ren ξ := ξ >>> ids`. The shift renaming $\uparrow$ is written `(+1)`.

As mentioned before, we implement the instantiation operation by first specializing to renamings.

```
Fixpoint rename (ξ : nat → nat) (s : term) : term :=
  match s with
   | Var x ⇒ Var (ξ x)
   | App s t ⇒ App (rename ξ s) (rename ξ t)
   | Lam s ⇒ Lam (rename (0 .: ξ >>> (+1)))
  end.
```

Using `rename`, we now define `up σ := ids 0 .: σ >>> rename (+1)`. Finally, using `up`, we define the full instantiation operation on terms.

```
Fixpoint inst (σ : nat → term) (s : term) : term :=
  match s with
   | Var x ⇒ σ x
   | App s t ⇒ App (inst σ s) (inst σ t)
   | Lam s ⇒ Lam (inst (up σ) s)
  end.
```

With instantiation, we define substitution composition `σ >> τ` as `σ >>> inst τ`. We write `s.[σ]` for $s[\sigma]$.

To complete the definition of instantiation we need to show that `rename` is a special case of `inst`. Concretely we must have `rename ξ s = inst (ren ξ) s` for all renamings `ξ` and terms `s`. The proof proceeds by induction on `s` and allows us to forget about `rename` in the remainder.

In order to show that the definitions above yield a model of the σ-calculus, we have to show every equation in Figure 1. Note however, that the only definitions that are specific to the term language at hand are the identity substitution, and the definition of instantiation. In order to show that our definitions yield a model of the σ-calculus, we only need to know that instantiation and the identity substitution behave correctly. For this, it suffices to establish

$$\begin{aligned} \mathsf{id}(x)[\sigma] &= \sigma(x) \\ s[\mathsf{id}] &= s \\ s[\sigma][\tau] &= s[\sigma \circ \tau] \end{aligned}$$

It is easy to check that given these three equations all the other equations in Figure 1 follow without any other assumptions about id or instantiation.

In the particular case of the untyped λ-calculus, the first equation $\mathsf{id}(x)[\sigma] = \sigma(x)$ holds by definition, while the second follows by a straightforward term induction. The third equation is more interesting, since the proof has to follow the inductive structure of the instantiation operation.

Formally, the proof proceeds in three steps. First we show $s[\xi][\tau] = s[\xi \circ \tau]$, by induction on s. Using this result we can show $s[\sigma][\xi] = s[\sigma \circ \xi]$, again by induction on s. We finally show the full equation, with another term induction. This proof boils down to showing that $\Uparrow\sigma \circ \Uparrow\tau = \Uparrow(\sigma \circ \tau)$, which depends on both specializations.

2.4 Realization in Autosubst

The library AUTOSUBST generalizes and automates the constructions from the previous sections for arbitrary term languages. A term language is specified as an inductive type with annotations.

```
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 2: Substitution Lemmas in `SubstLemmas`

For the untyped λ-calculus, we define the type of terms as follows.

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

Every term language must contain exactly one constructor for de Bruijn indices, i.e., a constructor which takes a single argument of type `var`. As before, this constructor serves as the identity substitution. The type `var` is an alias for `nat`.

Additionally, all binders must be marked. In this example, `Lam` is the only binder. It introduces a new index into the scope of its argument `s`. The annotation `{bind term}` is definitionally equal to `term`.

Using this definition of terms, we can derive the instantiation operation by defining an instance of the `Subst` type class using the tactic `derive`. This is comparable to the usage of deriving-clauses in the programming language Haskell. In addition, we need to define instances for the two auxiliary type classes `Ids` and `Rename`, defining the identity substitution and the instantiation of renamings.

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

Next, we derive the substitution lemmas from Figure 1 by generating an instance of the `SubstLemmas` type class.

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

This class contains the lemmas depicted in Figure 2.

These are all the necessary definitions to start using the library. In proofs involving terms and substitutions we can now benefit from the `autosubst` and `asimpl` tactics. The `autosubst` tactic is a decision procedure for equations between substitution expressions, that is, terms or substitutions build with the substitution operations. The tactic `asimpl` normalizes substitution expressions which appear in the goal. There are also focused variants, `asimpl in H` and `asimpl in *`, for simplifying a specific assumption `H` or all assumptions as well as the goal.

In order to use these tactics effectively, it is sometimes necessary to produce equational goals by hand. For instance, consider the following constructor for β-reduction.

```
step_beta s t : step (App (Lam s) t) s.[t .: ids]
```

This constructor can only be applied if the goal contains a term which is **syntactically** of the shape `s.[t .: ids]`. In order to apply the constructor to a goal of the form `step (App (Lam s) t) u`, we need a lemma with an equational hypothesis.

```
step_ebeta s u t : u = s.[t .: ids] → step (App (Lam s) t) u
```

We can then apply `step_ebeta` and show the subgoal using `autosubst`. Alternatively, we could have used `step_ebeta` directly in the definition of the reduction relation. With this design choice, we can show simple properties like substitutivity of reduction with an almost trivial proof script.

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

The tactic `autosubst` solves the crucial substitution lemma

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

Similar considerations apply to every lemma statement involving specific substitutions.

Internally, AUTOSUBST differs from a straightforward implementation of the σ-calculus in several ways. The σ-calculus contains a number of equations which are specific to the untyped λ-calculus. If we wanted to use the same approach for a new term language, we would need to extend the rewriting system with equations specific to the new term language. Instead, we construct a definition of instantiation with the appropriate simplification behavior. The automation tactics use a combination of rewriting and term simplification.

Internally, instantiation is defined in terms of renaming. We hide this fact by treating `up` as if it was an additional primitive. The automation tactics work by unfolding these definitions and rewriting substitution expressions to an internal normal form. In the case of `asimpl`, we then fold the expressions back into a more readable format.

AUTOSUBST is completely written in Coq. We synthesize the substitution operations using the meta-programming capabilities of Ltac.

3 Heterogeneous Substitutions

So far, we have only considered single-sorted syntactic theories. However, many practical programming languages distinguish at least between terms and types. In such a setting, we may have more than one substitution operations on terms, which may interact with one another.

As before, we shall introduce the necessary machinery with a concrete example. In this section we consider a two-sorted presentation of System F.

$$\begin{aligned} A, B &::= X \mid A \to B \mid \forall. A \\ s, t &::= x \mid s\, t \mid \lambda A.\, s \mid \Lambda. s \mid s\, A \end{aligned}$$

Substituting types in types works as before. Term substitutions are complicated by the fact that terms contain type binders.

$$\begin{aligned} x[\sigma] &= \sigma(x) \\ (s\, t)[\sigma] &= s[\sigma]\, t[\sigma] \\ (\lambda A.\, s)[\sigma] &= \lambda A.\, s[\Uparrow \sigma] \end{aligned} \qquad \begin{aligned} (s\, A)[\sigma] &= s[\sigma]\, A \\ (\Lambda. s)[\sigma] &= \Lambda. s[\sigma \bullet \uparrow] \end{aligned}$$

Upon traversing a type binder, we need to increment all type variables in σ. In order to substitute types in terms, we introduce heterogeneous instantiation and composition operations ($s\|[\theta]$ and $\sigma \bullet \theta$). To avoid confusion with term substitutions, we will write θ, θ' for type substitutions.

$$\begin{aligned} x\|[\theta] &= x \\ (s\, t)\|[\theta] &= s\|[\theta]\, t\|[\theta] \\ (\lambda A.\, s)\|[\theta] &= \lambda A[\theta].\, s\|[\theta] \end{aligned} \qquad \begin{aligned} (s\, A)\|[\theta] &= s\|[\theta]\, A[\theta] \\ (\Lambda. s)\|[\theta] &= \Lambda. s\|[\Uparrow \theta] \\ (\sigma \bullet \theta)(x) &= \sigma(x)\|[\theta] \end{aligned}$$

We need a number of lemmas about heterogeneous instantiation in order to work with it using equational reasoning. The situation is analogous to the case of ordinary instantiation, expect that heterogeneous substitutions have to be invariant in the image of the identity substitution on terms.

$$\begin{aligned} \mathsf{id}(x)\|[\theta] &= \mathsf{id}(x) \\ s\|[\mathsf{id}] &= s \\ s\|[\theta]\|[\theta]' &= s\|[\theta \circ \theta'] \end{aligned}$$

Note that the identity substitution in the first equation is the identity substitution on terms, while in the second equation, it refers to the identity substitution on types.

Furthermore, in order to show the substitution lemmas for terms, we need to know something about the interaction between the two kinds of substitutions. The fact to take care of is that terms may contain types, but types do not contain terms. Thus, we can push a type substitution under a term substitution, so long as we take care that the type substitution was applied in the image of the term substitution.

$$s[\sigma]\|[\theta] = s\|[\theta][\sigma \bullet \theta]$$

4 Case Studies

We have used AUTOSUBST to mechanize proofs about the metatheory of System F, System $F_{<:}$, and CC_ω. In each case study we define an inductive type of terms and derive the substitution operations using AUTOSUBST. In the case of System F and System $F_{<:}$, this uses AUTOSUBST's support for heterogeneous substitutions as presented in section 3.

For the proofs, we follow the same general strategy as presented in subsection 2.2. Whenever we encounter a lemma concerning a specific substitution, we try to find a generalization over all substitutions. For instance, in proofs of type preservation, we must show the admissibility of β-substitution for the typing relation. We generalize and show the admissibility of every "well-typed" substitution.

$$\frac{\Gamma, A \vdash s : B \qquad \Gamma \vdash t : A}{\Gamma \vdash s[t \cdot \mathsf{id}] : B} \rightsquigarrow \frac{\Gamma \vdash s : A \qquad \sigma : \Delta \to \Gamma}{\Delta \vdash s[\sigma] : A}$$

where $\sigma : \Delta \to \Gamma$ means that every $\sigma(x)$ is a term of type Γ_x in context Δ. However, the details of this definition depend on the typing relation. In the literature, such well-typed substitutions are known as **context morphisms** [9].

Context morphisms are more powerful than the β-substitution lemma. In particular, we obtain weakening by showing $\uparrow : \Gamma, A \to \Gamma$ and the admissibility of β by showing that if $\Gamma \vdash s : A$, then $s \cdot \mathsf{id} : \Gamma \to \Gamma, A$.

The proof of a context morphism lemma always follows the inductive structure of the instantiation operation. We first show the lemma for renamings. Then using the lemma for renamings, we show that $\Uparrow$ corresponds to context extension, that is, $\Uparrow\sigma : \Delta, A \to \Gamma, A$, whenever $\sigma : \Delta \to \Gamma$. From this, the full context morphism lemma follows.

In every case study, we show all equational substitution lemmas automatically with AUTOSUBST. With the correct definitions and lemma statements, most proofs are straightforward.

4.1 Type Preservation for CC_ω

Type preservation of CC_ω boils down to a number of inversion lemmas for typing derivations, as well as the admissibility of substitution. The former depend on the Church-Rosser theorem, whose proof proceeds exactly as illustrated in subsection 2.2.

The main difference between a normal paper presentation of CC_ω and the de Bruijn presentation is in the case of the variable rule. Context lookup is needed to implement the variable rule.

$$\frac{(x, A) \in \Gamma}{\Gamma \vdash x : A} \rightsquigarrow \frac{x < |\Gamma|}{\Gamma \vdash x : \Gamma_x}$$

In the non-dependent case, we can implement Γ_x by list lookup. In this case, however, the context is dependent. For dependent contexts Γ, we need to ensure that Γ_x is a valid type in Γ. In Coq, we totalize this function by adding a case for the empty context.

$$\begin{aligned}(\Gamma, A)_0 &= A[\uparrow]\\ (\Gamma, A)_{x+1} &= \Gamma_x[\uparrow]\end{aligned}$$

The context morphism lemma for CC_ω takes the form

$$\frac{\Gamma \vdash s : A \qquad \sigma : \Delta \to \Gamma}{\Delta \vdash s[\sigma] : A[\sigma]} \qquad \sigma : \Delta \to \Gamma \;:=\; \forall x < |\Gamma|, \Delta \vdash \sigma(x) : \Gamma_x[\sigma]$$

Apart from the obvious instances for weakening and β, we also obtain a context conversion lemma as a corollary. If we have two types $A \equiv B$, then $\mathsf{id} : \Gamma, B \to \Gamma, A$. Instantiating the context morphism lemma yields

$$\frac{\Gamma, A \vdash s : C \qquad A \equiv B}{\Gamma, B \vdash s : C}$$

4.2 Strong Normalization for System F

We follow Girard's proof of strong normalization [8]. The proof consists of defining a type-indexed family of sets of terms $L(A)$ such that $s \in L(A)$ implies that s is strongly normalizing. We then show that every term of type A is contained in $L(A)$, which implies the strong normalization result. The main technical difficulty with this proof is that we need to generalize the latter statement. Instead of showing that $\Gamma \vdash s : A$ implies $s \in L(A)$, we need to show that $s[\sigma] \in L(A)$ for all substitutions σ such that $\sigma(x) \in L(\Gamma_x)$. This generalization is standard and extensively documented in the literature.[1] In the proof, we have some occasion to generalize lemmas over all substitutions, but otherwise there are no surprises.

We still need to show how to define System F using de Bruijn terms.

In the named presentation of System F, we have two contexts, a context Δ for type variables and a context Γ for term variables. The rule for type abstraction is usually rendered with a side condition.

$$\frac{\Delta, X; \Gamma \vdash s : A}{\Delta; \Gamma \vdash \Lambda X.\, s : \forall X.\, A} \quad (X \notin \mathrm{FV}(\Gamma))$$

For de Bruijn terms, we can drop the context Δ, since it only asserts that at most a certain number of type variables are free. This constraint is orthogonal to the

[1]There are a large number of details which are swept under the carpet in this short overview. The formalization tells the whole story.

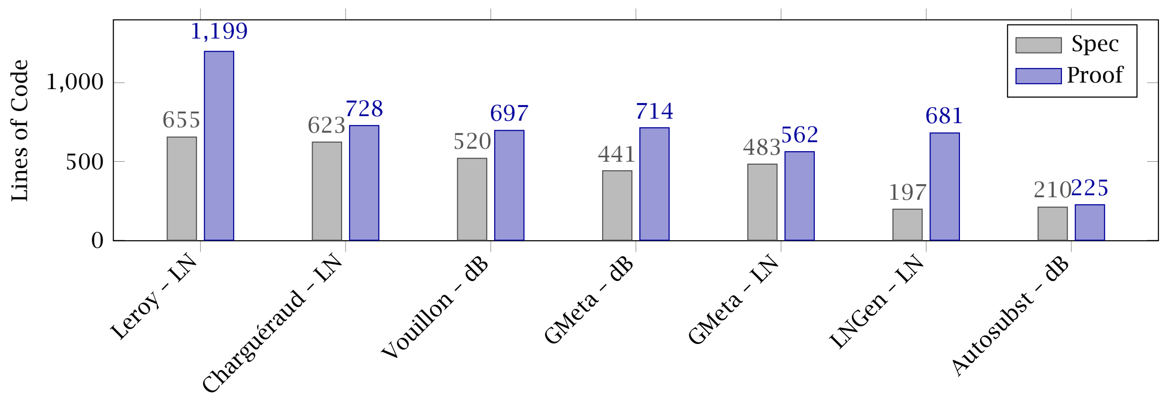

Figure 3: Comparison of Coq solutions to part A of the POPLmark challenge

typing relation. The freshness assumption can in turn be implemented similar to η-reduction. Note that we do not abstract over an arbitrary variable X, but rather over the index 0. If Γ is an arbitrary context, then 0 is not free in the context $\Gamma[\uparrow]$, where we instantiate every type in Γ under $\uparrow$. This leads to the following concise definition.

$$\frac{\Gamma[\uparrow] \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

All other inference rules are straightforward.

4.3 POPLmark Challenge

The POPLmark [4] is a benchmark to "measur[e] progress [...] in mechanizing the metatheory of programming languages". We solve part A, which concerns progress and preservation of System $F_{<:}$.

The typing relation of System $F_{<:}$ carries two contexts: A context Δ with subtyping assumptions for type variables and a context Γ with typing assumptions for terms. The context Δ is dependent, while Γ is non-dependent.

The proof of type preservation decomposes into a sequence of context morphism lemmas. In part 1A of the POPLmark challenge, we show the context morphism lemma for subtyping. For part 2A, we prove similar context morphism lemmas for the typing relation. These proofs are all simple and regular. The other lemmas are all very close to the informal proof.

In Figure 3, we compare lines of code using `coqwc`, excluding parts that the authors marked as reusable libraries.

- Xavier Leroy's solution[2] is a self-contained development using the locally nameless representation.
- Arthur Charguéraud's solution[3] also uses the locally nameless representation, but uses a small library to reason about the locally nameless representation.
- Jérôme Vouillon[4] uses de Bruijn indices in a self-contained development.
- There are two solutions using the GMeta [10] library. One using locally nameless and one using de Bruijn indices.
- There is a solution[5] using the LNGen [5] library, which generates substitution operations for locally nameless terms and proofs for the corresponding infrastructure lemmas.

5 Conclusion

There is an abundance of proposals for the formalization of binders: named syntax with α-equivalence and freshness-assumptions, de Bruijn indices, the locally nameless representation, higher-order abstract syntax, and nominal logic to name just a few contenders. The POPLmark Challenge [4] has been set up as a benchmark for these approaches and to encourage further development of solutions that work and scale.

However, the right method to choose depends on a lot of factors, the most important of which is probably the logic or theorem prover that one is working with. For example, higher-order abstract syntax and nominal logic require special support from the meta-logic in most variations. On the other hand, pure first-order approaches like named syntax, de Bruijn indices and the locally nameless representation work in most systems.

Another important distinction is the distance to paper proofs. Here, approaches with named contexts like named terms and locally nameless are closer, at the price of having to deal with issues of freshness manually. Our approach is rather distant from the usual paper proofs, which tend to use single-variable substitutions. In this respect, it is more distant than the more common approach using single-variable de Bruijn substitutions. On the other hand, our approach removes most of the technicalities of single-variable de Bruijn substitutions. Our decision procedure relies crucially on the fact that we use parallel substitutions. We believe that the elegance of this pure and parallel de Bruijn style is worth the paradigm shift.

[2] `http://www.seas.upenn.edu/~plclub/poplmark/leroy.html`
[3] `http://www.chargueraud.org/softs/ln/`
[4] `http://www.seas.upenn.edu/~plclub/poplmark/vouillon.html`
[5] `http://www.cis.upenn.edu/~sweirich/papers/lngen/`

5.1 Future Work

We identified the following shortcomings with the current implementation of AUTOSUBST, which we would like to address in future work.

- The current approach to heterogeneous substitutions is rather ad-hoc. It is not extensible to mutually recursive types and it is lacking a completeness result.
- AUTOSUBST lacks support for binders with variable arity. This is needed to formalize pattern matching, as used in part B of the POPLmark challenge.
- AUTOSUBST generates definitions using Ltac, which is rather fragile. Proper error reporting is currently not available, and in some cases it is necessary to manually inspect the generated code for errors. It seems to be almost impossible to build an Ltac script that is reliable in corner cases, due to the unpredictable nature of the Ltac semantics. Additionally, AUTOSUBST cannot handle mutually recursive term types. These problems could be solved by writing the code generator in an external programming language, or by extending the metaprogramming capabilities of Coq.

References

[1] Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming 1(4), 375–416 (1991)

[2] Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs, Lecture Notes in Computer Science, vol. 3839, pp. 1–16. Springer Berlin Heidelberg (2006)

[3] Anand, A., Rahli, V.: A generic approach to proofs about substitution. In: Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice. p. 5. ACM (2014)

[4] Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark challenge. In: Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 3603, pp. 50–65. Springer Berlin Heidelberg (2005)

[5] Aydemir, B.E., Weirich, S.: LNgen: Tool support for locally nameless representations. Tech. rep., University of Pennsylvania (2010)

[6] de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) 75(5), 381 – 392 (1972)

[7] Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ACM Sigplan Notices. vol. 43, pp. 143–156. ACM (2008)

[8] Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and types, vol. 7. Cambridge University Press Cambridge (1989)

[9] Goguen, H., McKinna, J.: Candidates for substitution. LFCS report series - Laboratory for Foundations of Computer Science ECS LFCS (1997)

[10] Lee, G., Oliveira, B.C., Cho, S., Yi, K.: GMeta: A generic formal metatheory framework for first-order representations. In: Programming Languages and Systems, Lecture Notes in Computer Science, vol. 7211, pp. 436–455. Springer Berlin Heidelberg (2012)

[11] Martin-Löf, P.: An intuitionistic theory of types. Twenty-five years of constructive type theory 36, 127–172 (1998)

[12] Polonowski, E.: Automatically generated infrastructure for de Bruijn syntaxes. In: Interactive Theorem Proving, Lecture Notes in Computer Science, vol. 7998, pp. 402–417. Springer Berlin Heidelberg (2013)

[13] Pottier, F.: DBLIB, a Coq library for dealing with binding using de Bruijn indices. `https://github.com/fpottier/dblib` (Dec 2013)

[14] Schäfer, S., Smolka, G., Tebbi, T.: Completeness and decidability of de bruijn substitution algebra in coq. In: Proceedings of the 2015 Conference on Certified Programs and Proofs. pp. 67–73. CPP '15, ACM, New York, NY, USA (Jan 2015)

[15] Schäfer, S., Tebbi, T.: Autosubst: Automation for de Bruijn syntax and substitution in Coq. Available at `www.ps.uni-saarland.de/autosubst` (August 2014)

[16] Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. Journal of Functional Programming 20(1), 71 (2010)

[17] Takahashi, M.: Parallel reductions in λ-calculus. Information and computation 118(1), 120–127 (1995)