Autosubst 2: Towards Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions

Jonas Kaiser
Saarland University
Saarbrücken, Germany
jkaiser@ps.uni-saarland.de

Steven Schäfer
Saarland University
Saarbrücken, Germany
schaefer@ps.uni-saarland.de

Kathrin Stark
Saarland University
Saarbrücken, Germany
kstark@ps.uni-saarland.de

ABSTRACT

Formalising metatheory in the Coq proof assistant is tedious as native support for reasoning about languages with binders is marginal at best. The Autosubst framework [9] automates working with de Brujin terms: for each annotated inductive type, it generates a corresponding substitution operation and a decision procedure for assumption-free substitution lemmas. A key part of the equational theory are parallel substitutions which combine multiple single-variable substitutions. However, due to its separate treatment of sorts Autosubst lacks support for mutual inductive types. This restriction is removed in our prototype implementation of Autosubst 2: second-order HOAS specifications are translated into potentially mutual inductive term sorts. Again, parallelising substitutions is the key: we introduce vector substitutions to combine the application of multiple parallel substitutions into exactly one instantiation operation for each sort. The resulting equational theory is both simpler and more expressive than that of the original Autosubst framework.

CCS CONCEPTS

• **Theory of computation** → *Automated reasoning*; *Type theory*; *Operational semantics*;

KEYWORDS

de Bruijn representation, parallel substitutions, σ-calculus, multi-sorted terms

ACM Reference format:
Jonas Kaiser, Steven Schäfer, and Kathrin Stark. 2017. Autosubst 2: Towards Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions. In *Proceedings of LFMTP '17, Oxford, United Kingdom, September 8, 2017,* 5 pages.
https://doi.org/10.1145/3130261.3130263

1 INTRODUCTION

Formalising the metatheory of programming languages and logical systems in a proof assistant requires the treatment of syntax with binders. In systems without native support for such reasoning, like the general purpose proof assistant Coq, we have to rely on libraries to avoid heaps of boilerplate and uninteresting technicalities.

With Autosubst [9] we introduced such a library for de Bruijn syntax in Coq. Based on de Bruijn's original presentation [5] and the σ-calculus [1], Autosubst takes an annotated inductive type of terms, automatically derives a capture-avoiding instantiation operation for parallel substitutions, and provides a decision procedure for assumption-free substitution lemmas [8]. We have successfully used Autosubst in several case studies, ranging from strong normalisation proofs to the metatheory of Martin-Löf type theory [9], as well as equivalence proofs of alternative syntactic presentations of System F [6].

That said, there are several shortcomings with the current version of Autosubst:

(1) The generation of an instantiation operation for a given term sort, that is, a syntactic class, automatically equips the sort with a variable constructor.
(2) The handling of heterogeneous substitutions, i.e., multiple instantiation operations on a single term sort, is ad-hoc.
(3) The implementation fails to handle certain inductive term sorts accepted by Coq, in particular, due to the handling of heterogeneous substitutions, mutual inductive sorts.
(4) Although technically dispensable, the automation is tied to the axiom of functional extensionality.
(5) The implementation is simply too slow.

In this paper we report on a work-in-progress extension of Autosubst to address these problems.

The first three issues are solved by shifting the input language. Instead of accepting pre-existing inductive types in Coq as term sorts, we start from a higher-order abstract syntax (HOAS) [7] system specification. Based on the specification we compute which sorts require variables and which sorts have to be declared as mutually inductive. At the moment we only accept second-order specifications, that is, we do not admit HOAS constants like the μ-operator found in [2]: `mu : ((tm → nam) → nam) → tm`. As a result, the generated inductive types in Coq are simple in the sense that they do not have constructors accepting functions as arguments. Together with carefully chosen proof statements this allows us to reason about them without assuming the axiom of functional extensionality, thus alleviating the fourth issue.

The main contribution of this paper is our novel treatment of heterogeneous substitutions. Instead of equipping a given sort x with a separate instantiation operation for each sort y that may occur as a variable in x, we generate a single instantiation operation that takes a vector of parallel substitutions with one component for each occurring variable sort y. For sorts without any variable occurrences no instantiation is generated.

LFMTP '17, September 8, 2017, Oxford, United Kingdom

ACM ISBN 978-1-4503-5374-8/17/09…$15.00
https://doi.org/10.1145/3130261.3130263

Using vectors of parallel substitutions simplifies the equational theory of substitution lemmas in the heterogeneous setting. We extend the automation of Autosubst accordingly, using a straightforward extension of the σ-calculus. In contrast to Autosubst, we generate the automation via an external tool, using a type-class based simplification mechanism to drop the axiom of functional extensionality and meet our performance goals in the future.

To demonstrate the benefit of mutual recursive types with heterogeneous substitutions we revisit a case study from [9]. We show weak normalisation of call-by-value System F, hereafter called $\mathrm{F_{CBV}}$, by making a syntactic distinction between terms and values. This syntactic distinction simplifies the definitions and leads to an extremely short proof. All emerging substitution lemmas are automatically solved by our extended automation tactic.

A first prototype of Autosubst 2 and the accompanying material are available at: https://www.ps.uni-saarland.de/extras/lfmtp17.

2 FROM PARALLEL TO VECTOR SUBSTITUTIONS

The main feature of de Bruijn syntax is the absence of variable names. Variables are instead represented as numerical indices, where n references the n-th enclosing binder of the corresponding scope. The following grammar gives System F in de Bruijn representation, where we distinguish terms and values.

$$\begin{array}{rcll} A, B \in ty & ::= & X \mid A \to B \mid \forall.\, A & \text{Types} \\ s, t \in tm & ::= & s\,t \mid s\,A \mid v & \text{Terms} \\ u, v \in vl & ::= & x \mid \lambda A.\, s \mid \Lambda.\, s & \text{Values} \end{array}$$

We recall the definition of instantiating a type A with a parallel type substitution $\sigma : \mathbb{N} \to ty$, written $A[\sigma]$. The substitution acts on all free type variable in A at once. We define $A[\sigma]$ mutually recursive with the forward composition of substitutions:

$$\begin{array}{rclrcl} X[\sigma] & = & \sigma\, X & (\sigma_1 \circ \sigma_2)\, X & = & (\sigma_1\, X)[\sigma_2] \\ (A \to B)[\sigma] & = & A[\sigma] \to B[\sigma] & & & \\ (\forall.\, A)[\sigma] & = & \forall.\, A[\Uparrow^{ty}_{ty}\, \sigma] & \text{with } \Uparrow^{ty}_{ty}\, \sigma & = & 0_{ty} \cdot \sigma \circ \uparrow \end{array}$$

The substitution $A \cdot \sigma$ maps the index 0 to A and indices $n + 1$ to $\sigma\, n$. Note that this *stream cons* operation binds weaker than composition. The *shift* substitution $\uparrow$ simply maps every index n to $n + 1$.

The beauty of this design over single-point substitutions lies in the fact that multiple single-point substitutions interfere with each other and permuting them introduces non-trivial side conditions. Combining them into a parallel substitution leads to a more uniform treatment and is crucial for an elegant equational theory.

Note that the mutual recursion above is not structural. In the formalisation we follow the pattern presented in [3, 9] and first develop the setup for the special case of renamings (substitutions which only substitute variables).

Let us next consider terms and values. As both types and values can appear as subexpressions, so do type and value variables and we have to be able to substitute for both.

One possible solution is to equip terms and values with two instantiation operations each – one for each occurring sort with variables. This is exactly what was done in Autosubst, at least for a limited class of syntactic systems. We again face the problem that the various instantiations interfere, and permuting them is non-trivial. Take for example

$$s[\tau]_{vl}[\sigma]_{ty} = s[\sigma]_{ty}[\lambda x.\, (\sigma\, x)[\tau]_{ty}]_{vl}$$

where permuting the two substitutions requires us to replace types in substituted values. Even more important, this fails to scale to mutual inductive sorts like those of our example $\mathrm{F_{CBV}}$.

$$\begin{array}{rclrcl} (s\,t)[\sigma, \tau] & = & s[\sigma, \tau]\, t[\sigma, \tau] & x[\sigma, \tau] & = & \tau\, x \\ (s\,A)[\sigma, \tau] & = & s[\sigma, \tau]\, A[\sigma] & (\lambda A.\, s)[\sigma, \tau] & = & \lambda A[\sigma].\, s[\Uparrow^{vl}_{tm}\, (\sigma, \tau)] \\ & & & (\Lambda.\, s)[\sigma, \tau] & = & \Lambda.\, s[\Uparrow^{ty}_{tm}\, (\sigma, \tau)] \end{array}$$

$$\begin{aligned} \Uparrow^{vl}_{tm}\, (\sigma, \tau) &= (\sigma, 0_{vl} \cdot \tau \circ (\mathsf{id}_{ty}, \uparrow)) \\ \Uparrow^{ty}_{tm}\, (\sigma, \tau) &= (0_{ty} \cdot \sigma \circ \uparrow, \tau \circ (\uparrow, \mathsf{id}_{vl})) \\ (\tau \circ (\sigma', \tau'))\, x &= (\tau\, x)[\sigma', \tau'] \end{aligned}$$

Figure 1: Term and value substitutions for $\mathrm{F_{CBV}}$.

Again, parallelising substitutions is the key. Just as we combined several single-point substitutions into a parallel substitution, we now combine multiple parallel substitutions into a single vector of substitutions, with one component for each sort that may occur in a variable position. We will see that this leads again to a more uniform treatment and a simple equational theory. In the following we give the required definitions for $\mathrm{F_{CBV}}$ to illustrate the approach.

The instantiation operations for terms and values are defined in Figure 1, again mutually recursive with the forward composition operation. We write $s[\sigma, \tau]$ for a term s where all type variables are substituted according to σ and all value variables according to τ, and similarly for values. The following aspects are worth pointing out.

First, whenever we reach a variable, we have to project the correct component, e.g. $x[\sigma, \tau] = \tau\, x$ for value variables.

Second, when a given subterm is of a different sort, we have to select the correct instantiation function and subvector. Take for example $(s\,A)[\sigma, \tau]$, where the correct subvector for instantiating the subterm A is $[\sigma]$.

Third, and most interesting, the traversal of binders changes the interpretation of indices in scope. We have to adjust the full substitution vector via an *up-operation* which is more involved than in the single-sorted setting (cf. $\Uparrow^{ty}_{ty}$). The component that corresponds to the sort of the binder we just traversed, say σ, is modified almost as before. While the index 0 is mapped to 0 as usual, we have to ensure that $n + 1$ is first mapped to σn and then adjusted to bypass the new binder. For types this was achieved by simply postcomposing $\uparrow$ to σ. We now instead have to postcompose a vector substitution which matches the codomain of σ, has a shift for the bound sort, and is otherwise the identity. We further have to construct and postcompose such adjustments to all other components σ' of the original vector substitution. For our concrete example these are the two operations $\Uparrow^{vl}_{tm}$ and $\Uparrow^{ty}_{tm}$ defined in Figure 1 which both construct substitutions suitable for the sort of terms while incorporating a newly bound value or, respectively, type. When we observe their uses carefully, we see that they act on substitutions for the sort of

$$
\begin{aligned}
(s \cdot \sigma)\, 0 &= s & \uparrow \circ\, s \cdot \sigma &\equiv \sigma \\
(s \cdot \sigma)\, (n+1) &= \sigma\, n & \sigma\, 0 \cdot \uparrow \circ\, \sigma &\equiv \sigma
\end{aligned}
$$

$$(s \cdot \sigma') \circ \overline{\sigma} \equiv s[\overline{\sigma}] \cdot \sigma' \circ \overline{\sigma}$$

Figure 2: Interplay of cons, composition, and shifting.

$$
\begin{aligned}
s[\mathrm{id}_{ty}, \mathrm{id}_{vl}] &= s & (\dagger) \\
s[\sigma, \tau][\sigma', \tau'] &= s[\sigma \circ \sigma', \tau \circ (\sigma', \tau')] & (\dagger) \\
\mathrm{id}_{vl} \circ (\sigma, \tau) &\equiv \tau \\
\tau \circ (\mathrm{id}_{ty}, \mathrm{id}_{vl}) &\equiv \tau \\
(\tau \circ (\sigma', \tau')) \circ (\sigma'', \tau'') &\equiv \tau \circ (\sigma' \circ \sigma'', \tau' \circ (\sigma'', \tau''))
\end{aligned}
$$

Figure 3: Equational rules for values.

values, indicating that the subvector cast mentioned above may in fact be the identity. We further observe that the postcomposed adjustment may itself not have a component for the bound sort, in which case the adjustment degenerates to the identity everywhere and is tacitly omitted. Consider for example the definition of $\Uparrow^{vl}_{tm}$, which does not adjust the type component of the substitution, as values do not occur in types.

Based on the aforementioned aspects we can extend the σ-calculus [1] to vector substitutions, and thus obtain an elegant equational theory. We recall the rules that govern the interplay of basic forward composition, cons, and shifting in Figure 2, where $\equiv$ denotes extensional equality of substitutions, lifted pointwise to vectors. These hold regardless of the concrete syntactic system, like F_{CBV}.

The last equivalence of Figure 2 is different, as it describes a family of equivalences with one instance for each occurring shape of vector substitutions. This is tied to the fact that for each shape of vector substitutions we obtain a separate composition operation that precomposes a simple substitution to vector substitutions of this shape. We use $\overline{\sigma}$ to denote arbitrary vector substitutions.

In addition, we have rules that are specific to each syntactic sort, as they depend on the shape of the corresponding vector substitution and instantiation operation. We give the rules for the values of F_{CBV} in Figure 3. The rules describe the interplay of identities, composition, and instantiations with vector substitutions. We further obtain, for each sort, an extensionality principle that connects the equivalence of two substitutions to the equality of terms under these substitutions. For the values of F_{CBV} we have, for example,

$$(\sigma, \tau) \equiv (\sigma', \tau') \implies s[\sigma, \tau] = s[\sigma', \tau']. \qquad (\dagger)$$

If we combine the general rules of Figure 2, the sort specific rules for each sort, and the defining equations of instantiation and composition and read them from left to right we obtain a rewriting system that can be used to simplify instantiation expressions. This is exploited by our simplification tactic *asimpl*.

```
ty, tm, vl : Type
arr : ty → ty → ty
all : (ty → ty) → ty

app : tm → tm → tm
tapp : tm → ty → tm
vt   : vl → tm

lam  : ty → (vl → tm) → vl
tlam : (ty → tm) → vl
```

```
Inductive ty : Type :=
  | var_ty : index → ty
  | arr : ty → ty → ty
  | all : ty → ty.

Inductive tm : Type :=
  | app : tm → tm → tm
  | tapp : tm → ty → tm
  | vt : vl → tm
 with vl : Type :=
  | var_vl : index → vl
  | lam : ty → tm → vl
  | tlam : tm → vl.
```

Figure 4: HOAS specification of F_{CBV} (left) and the corresponding inductively defined de Bruijn sorts (right).

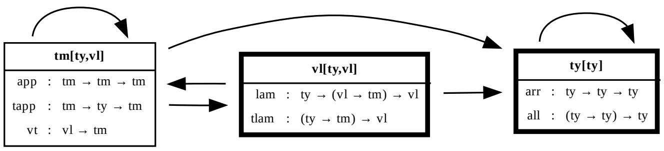

Figure 5: Dependency graph of F_{CBV}.

For each sort we identify three rules of the rewriting system as key lemmas, marked using (†), that suffice to obtain all remaining rules for the corresponding sort. These have to be established from first principles, following the structure of instantiation.

3 FROM HOAS TO DE BRUIJN

All definitions and statements of the previous section follow a regular pattern where the only real input was the grammar of F_{CBV}. We exploit this regularity and automatically generate the inductive term sorts, the corresponding vector instantiation operations, and the equational theory for a given concise syntax description.

Our prototype implementation in Haskell parses a TWELF-like second order HOAS system specification and produces the desired output as a plain Coq source file. A sample input specification and the desired inductive term sorts for F_{CBV} are shown in Figure 4.

To understand why such HOAS specifications suffice to generate the wealth of structure outlined above, we need to study the notion of direct occurrence, a relation on syntactic sorts. Given a HOAS constructor, say `lam : ty → (vl → tm) → vl`, we will refer to the result type of each argument as *head* of said argument, here `ty` and `tm`. When a given argument, e.g. `vl → tm`, has premises, we will call them the *binders* of the argument, here `vl`. A sort y *occurs directly* in sort x exactly when it appears as an argument head in one of x's constructors. We refer to the transitive closure of direct occurence simply as *occurrence*.

At this point we can determine if a given sort has to be equipped with a variable constructor, as these are left implicit in the HOAS specification. A sort x requires a variable constructor iff x is a binder of some sort y and also occurs in y. For F_{CBV} this applies to `ty` and `vl`, but not to `tm`. If only the first condition is satisfied, the respective binding constructor is vacuous and our implementation produces a warning.

$$\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\, t : B} \qquad \frac{\Gamma \vdash s : \forall.\ A}{\Gamma \vdash s\, B : A[B \cdot \mathsf{id}_{ty}]} \qquad \frac{\Gamma \vdash^{v} v : A}{\Gamma \vdash v : A}$$

$$\frac{x < |\Gamma|}{\Gamma \vdash^{v} \Gamma_x : A} \qquad \frac{\Gamma, A \vdash s : B}{\Gamma \vdash^{v} \lambda A.\ s : A \to B} \qquad \frac{\Gamma[\uparrow] \vdash s : A}{\Gamma \vdash^{v} \Lambda.\ s : \forall.\ A}$$

$$\frac{s \Downarrow \lambda A.\ b \qquad t \Downarrow u \qquad b[\mathsf{id}_{ty}, u \cdot \mathsf{id}_{vl}] \Downarrow v}{s\, t \Downarrow v} \qquad \frac{s \Downarrow \Lambda.\ b \qquad b[A \cdot \mathsf{id}_{ty}, \mathsf{id}_{vl}] \Downarrow v}{s\, A \Downarrow v} \qquad \frac{}{v \Downarrow v}$$

Figure 6: Type system and reduction relation of $\mathrm{F_{CBV}}$.

The information can be visualised as a directed dependency graph, where nodes correspond to sorts and an edge from x to y indicates the direct occurrence of y in x. Sorts that require variables are marked by a bold border. The dependency graph for $\mathrm{F_{CBV}}$ is shown in Figure 5. We also show the shape of the corresponding vector substitutions, that is a list of sorts that are the codomains for each required substitution component. To be precise, a vector substitution for a sort x must have a component for each occurring sort y which has variables. Here, `ty` requires only one component for `ty` itself, while `tm` and `vl` each require components for both `ty` and `vl`.

We now process this dependency graph in topological order, preserving the input order of sorts and constructors as much as possible, to generate the desired output. Care has to be taken, as sorts of a strongly connected component have to be processed simultaneously. This means that the corresponding inductive term sorts will be declared as mutually inductive, instantiation operations will be defined mutually recursive, and the equational rules of the affected sorts are proven simultaneously.

The generation of the inductive term sorts is straightforward. All we have to do is aggregate the constructors, strip binders and, if necessary, add a variable constructor. Instantiations are slightly more interesting, as this is where all the de Bruijn binding mechanisms are handled. As the shape of the substitution was determined from the *transitive* notion of occurrence, we know that a suitable subvector exists whenever we move into subexpressions of a different sort. For the correct choice of an up-operation we first determine the head of the binding argument x and then the bound sort y and then employ $\Uparrow_{x}^{y}$. The graph also tells us which of these up-operations have to be generated, and how.

For the key lemmas of the equational theory (marked by †) we construct explicit proof terms that follow the inductive structure of the term sorts. These Lemmas are then used to realise the aforementioned rewriting system under the tactic invocation *asimpl*.

4 WEAK NORMALISATION OF $\mathrm{F_{CBV}}$

To demonstrate our framework in action we present a concise formal proof that $\mathrm{F_{CBV}}$ is weakly normalising. In Figure 6, we define both the typing rules and a big-step reduction relation from terms to values in de Bruijn style.

We show that every closed, well-typed term s can be reduced to a value v, that is $s \Downarrow v$, using a unary logical relation. The logical relation interprets (open) types as mappings from environments to sets of values, realised as predicates. An environment ρ maps type variables to sets of values and we write $d \cdot \rho$ for ρ extended with a new type variable interpretation d.

Similar to the typing rules, the logical relation consists of two parts, a term interpretation $\llbracket A \rrbracket_\rho$ and a value interpretation $\llparenthesis A \rrparenthesis_\rho$.

$$\begin{aligned}
\llbracket A \rrbracket_\rho &:= \lambda s.\ \exists v.\ s \Downarrow v \land \llparenthesis A \rrparenthesis_\rho\, v \\
\llparenthesis X \rrparenthesis_\rho &:= \rho\, X \\
\llparenthesis A \to B \rrparenthesis_\rho &:= \{\, \lambda C.\ s \mid \forall v.\ \llparenthesis A \rrparenthesis_\rho\, v \to \llbracket B \rrbracket_\rho\, s[\mathsf{id}_{ty}, v \cdot \mathsf{id}_{vl}] \,\} \\
\llparenthesis \forall.\ A \rrparenthesis_\rho &:= \{\, \Lambda.\ s \mid \forall B d.\ \llbracket A \rrbracket_{d \cdot \rho}\, s[B \cdot \mathsf{id}_{ty}, \mathsf{id}_{vl}] \,\}
\end{aligned}$$

In order to handle type abstractions we need to know that this definition is compatible with type substitution and weakening.

Lemma 4.1. *For all types A, environments ρ, and renamings ξ we have $\llparenthesis A[\xi] \rrparenthesis_\rho = \llparenthesis A \rrparenthesis_{\xi \circ \rho}$. In particular, $\llparenthesis A[\uparrow] \rrparenthesis_{d \cdot \rho} = \llparenthesis A \rrparenthesis_\rho$ holds.*

Proof. By induction on A using the equations in Figure 2. □

Lemma 4.2. *For all types A, environments ρ, and substitutions σ we have $\llparenthesis A[\sigma] \rrparenthesis_\rho = \llparenthesis A \rrparenthesis_{\sigma \circ \llparenthesis - \rrparenthesis_\rho}$. The result trivially lifts to the term interpretation and we obtain $\llbracket A[B \cdot \mathsf{id}_{ty}] \rrbracket_\rho = \llbracket A \rrbracket_{\llparenthesis B \rrparenthesis_\rho \cdot \rho}$ as a special case.*

Proof. Induction on A using Lemma 4.1. □

We extend the value interpretation to terms in contexts and define semantic counterparts to our two syntactic typing relations.

$$\begin{aligned}
\llparenthesis \Gamma \rrparenthesis_\rho &:= \lambda \tau.\ \forall x < |\Gamma|.\ \llparenthesis \Gamma_x \rrparenthesis_\rho\, (\tau\, x) \\
\Gamma \vDash s : A &:= \forall \sigma \tau \rho.\ \llparenthesis \Gamma \rrparenthesis_\rho\, \tau \to \llbracket A \rrbracket_\rho\, s[\sigma, \tau] \\
\Gamma \vDash^{v} v : A &:= \forall \sigma \tau \rho.\ \llparenthesis \Gamma \rrparenthesis_\rho\, \tau \to \llparenthesis A \rrparenthesis_\rho\, v[\sigma, \tau]
\end{aligned}$$

We now prove that syntactic typing implies semantic typing.

Theorem 4.3 (Soundness). *For all Γ, s, v, A we have*

$$\Gamma \vdash s : A \to \Gamma \vDash s : A$$
$$\Gamma \vdash^{v} v : A \to \Gamma \vDash^{v} v : A$$

Proof. By mutual induction on the typing derivations. The type application case introduces a substitution on types which is handled with Lemma 4.2. Meanwhile type abstraction relies on Lemma 4.1. The proof also depends on two non-trivial substitution lemmas for the cases of abstraction and type abstraction.

$$s[\Uparrow_{tm}^{vl} (\sigma, \tau)][\mathsf{id}_{ty}, v \cdot \mathsf{id}_{vl}] = s[\sigma, v \cdot \tau]$$
$$s[\Uparrow_{tm}^{ty} (\sigma, \tau)][A \cdot \mathsf{id}_{ty}, \mathsf{id}_{vl}] = s[A \cdot \sigma, \tau]$$

Both are solved automatically by our framework. □

Corollary 4.4 (Weak Normalisation). *For all s, A we have*

$$\vdash s : A \ \to\ \exists v.\ s \Downarrow v$$

Note that our definitions and proofs rely on the syntactic distinction between terms and values. For the logical relation, it is crucial that ρ maps type variables to sets of values, instead of arbitrary terms. More details can be found in the accompanying formalisation.

5 CONCLUSION AND FUTURE WORK

We have outlined the theory and design of Autosubst 2, a Coq tool that supports reasoning about languages with binders for mutual inductive sorts. Given a HOAS specification, our tool generates a Coq source file containing inductive de Brujin term sorts, corresponding instantiation operations with vector substitutions, as well as type classes and instances that implement a normalisation procedure.

The presented implementation should be considered work in progress. The automation still requires further improvement and a systematic comparison with the previous approach. Following the ideas presented in [4], we further want to extend Autosubst to automatically prove results like Lemmas 4.1 and 4.2 which establish the compatibility of renaming and substitution of a given recursive definition over our term sorts.

At the moment Autosubst 2 is provided as an external tool but we are considering an implementation as a Coq plugin to provide a better user experience and further performance improvements.

REFERENCES

[1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. 1991. Explicit substitutions. *Journal of functional programming* 1, 4 (1991), 375–416.

[2] Andreas Abel. 2001. A Third-Order Representation of the $\lambda\mu$-Calculus. *Electronic Notes in Theoretical Computer Science* 58, 1 (2001), 97 – 114. MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding (in connection with IJCAR 2001).

[3] Robin Adams. 2004. Formalized metatheory with terms represented by an indexed family of types. In *International Workshop on Types for Proofs and Programs*. Springer, 1–16.

[4] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope Safe Programs and Their Proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. ACM, 195–207.

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

[6] Jonas Kaiser, Tobias Tebbi, and Gert Smolka. 2017. Equivalence of System F and $\lambda 2$ in Coq based on Context Morphism Lemmas. In *Proceedings of CPP 2017*. ACM.

[7] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. ACM, 199–208.

[8] Steven Schäfer, Gert Smolka, and Tobias Tebbi. 2015. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*. ACM, 67–73.

[9] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *International Conference on Interactive Theorem Proving*. Springer, 359–374.