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

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

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

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

Abstract

Formalising metatheory in the Coq proof assistant is tedious as reasoning with binders without native support requires a lot of uninteresting technicalities. To relieve users from so-produced boilerplate, the Autosubst framework automates working with de Bruijn terms: For each annotated inductive type, Autosubst generates a corresponding instantiation operation for parallel substitutions and a decision procedure for assumption-free substitution lemmas. However, Autosubst is implemented in Ltac, Coq's tactic language, and thus suffers from Ltac's limitations. In particular, Autosubst is restricted to Coq and unscoped, non-mutual inductive types with a single sort of variables. In this paper, we present a new version of Autosubst that overcomes these restrictions. Autosubst 2 is an external code generator, which translates second-order HOAS specifications into potentially mutual inductive term sorts. We extend the equational theory of Autosubst to the case of mutual inductive sorts by combining the application of multiple parallel substitutions into exactly one instantiation operation for each sort, i.e. we parallelise substitutions to vector substitutions. The resulting equational theory is both simpler and more expressive than that of the original Autosubst framework and allows us to present an even more elegant proof of part A of the POPLMark challenge.

CCS Concepts • **Theory of computation → Automated reasoning**; **Type theory**; **Operational semantics**; *Lambda calculus.*

Keywords de Bruijn representation, parallel substitutions, sigma-calculus, multi-sorted terms

CPP '19, January 14–15, 2019, Cascais, Portugal

ACM ISBN 978-1-4503-6222-1/19/01…$15.00
https://doi.org/10.1145/3293880.3294101

ACM Reference Format:
Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: Reasoning with Multi-sorted de Bruijn Terms and Vector Substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3293880.3294101

1 Introduction

Formalising the metatheory of programming languages and logical systems in a proof assistant requires the treatment of syntax with binders. However, using a system without native support for binders (like the general-purpose proof assistant Coq) requires a lot of boilerplate [Aydemir et al. 2005]. These uninteresting technicalities distract from the actual proofs and forgo the advantage of automation a proof assistant has.

There are multiple approaches to representing binders – de Bruijn [de Bruijn 1972], locally nameless [Aydemir et al. 2005], nominal sets [Pitts 2013], or HOAS [Pfenning and Elliott 1988]. While all approaches chase the common goal of simplifying proofs with binders, there are different tradeoffs and measures of success: How natural is it to work with? Which syntax can be handled? Which are the requirements for the logic of the proof assistant? How much boilerplate is necessary before and during use?

An approach that emphasizes the last problem and minimises the pure mechanisation effort is Autosubst [Schäfer et al. 2015b]. Autosubst is based on the σ-calculus [Abadi et al. 1991], a λ-calculus with explicit substitutions.

The σ-calculus handles substitutions by restricting itself to a small selection of substitution operations and combinators which are (1) still expressive enough to handle β- and η-reduction and (2) are closed under instantiation. These operations come with a terminating [Abadi et al. 1991], confluent [Curien et al. 1996], and complete [Schäfer et al. 2015a] rewriting system, which allows arguing about these operations in the form of a decision procedure for assumption-free substitution lemmas. Autosubst then forms a model of the σ-calculus with unscoped de Bruijn terms.

More specifically, given an annotated inductive type of terms, Autosubst automatically derives a model of an extended σ-calculus, comprising an instantiation operation and rewriting system. This means that we can generate every proof for a goal of the form $s = t$ which contains only syntactic expressions and instantiation – if it exists.

We have successfully used Autosubst in several case studies, ranging from strong normalisation proofs to the metatheory of Martin-Löf type theory [Schäfer et al. 2015b] and equivalence proofs of alternative syntactic presentations of System F [Kaiser et al. 2017b]. However, the derivation process is implemented in Ltac, Coq's tactic language. Ltac is no full programming language, and Autosubst 1 suffers from its limitations, namely:

1. The generation of an instantiation operation for a given syntactic class automatically equips the sort with a variable constructor.
2. Ltac is specific to Coq.
3. It is hard to maintain or extend Ltac code, e.g. to provide faster automation or automation without functional extensionality.
4. Ltac's semantics is non-dependent and allows no mutual definitions. Autosubst thus allows neither mutual inductive sorts nor well-scoped syntax. In general, it is not clear which exact class of syntax Autosubst can handle as Ltac tactics are used heuristically.
   As a consequence, the handling of heterogeneous substitutions, i.e., multiple instantiation operations on a single term sort, is ad-hoc.
5. It is difficult to extend Autosubst to more constructs than instantiation, e.g., syntax traversals [Allais et al. 2017; Kaiser et al. 2018].

In summary, Autosubst 1 cannot handle the syntax of a call-by-value variant of the lambda calculus, call-by-push value, or the π-calculus. The success in extending Autosubst 1 via Ltac is questionable – especially the lack of mutual recursion would require several work-arounds –, and, in the best case, error prone.

We thus propose a new implementation of Autosubst which amends the above-mentioned lack of flexibility and reliability and at the same time extends Autosubst's input language to mutual inductive sorts with multiple sorts of variables.

Our implementation comes in the form a code generator in three layers: It parses a Twelf-like second order HOAS system specification, analyses it, generates internal proof terms, and then produces the desired definitions and lemmas as a plain source file which can be read by the proof assistant.

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 as the μ-operator `mu : ((tm → nam) → nam) → tm` found in [Abel 2001]. As a result, the generated inductive types in Coq are simple in the sense that they do not have constructors accepting functions as arguments. The current version of Autosubst 2 is able to produce both unscoped and well-scoped Coq code. The extension to well-scoped syntax is technically straightforward, but gives the user a better check while writing definitions.

Moreover, Autosubst 1 chose to handle renamings – i.e., substitutions which only substitute variables, in our case not necessarily injective – as second-class: An unfortunate choice, as many statements require a proof for renamings first (Section 4). In our re-implementation, we introduce first-order renamings to Autosubst 2.

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.

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.

To demonstrate the benefit of mutual inductive types with heterogeneous substitutions we revisit a case study from [Schäfer et al. 2015b]. We show weak normalisation of call-by-value System F 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. We moreover present a more elegant proof of part A of the POPLMark challenge [Aydemir et al. 2005] using Autosubst 2 and shortly present other case studies enabled by Autosubst 2.

Contributions. This paper revisits the efforts reported in a previous work-in-progress paper [Kaiser et al. 2017a]. As such, it complements and extends the σ-calculus and Autosubst 1 to 1.) multivariate, 2.) mutual inductive, and 3.) well-scoped syntax. This includes an extension of both instantiation and the corresponding rewriting system.

Our re-implementation as an external tool moreover creates the basis for a more flexible tool to argue about substitutions. We moreover provide a proof of weak normalisation of call-by-value System F and a new, improved proof of the POPLMark challenge.

The Coq formalisation of all results in this paper and the Autosubst 2 tool itself are available online[1].

2 Preliminaries

The main feature of de Bruijn syntax is the absence of variable names. In well-scoped de Bruijn [Adams 2004], variables are instead represented as indices taken from a finite

[1] https://www.ps.uni-saarland.de/extras/autosubst2/

$$0 : \mathbb{I}_{n+1} \qquad \mathsf{id} : \mathbb{I}_n \to \mathbb{I}_n$$
$$\uparrow : \mathbb{I}_n \to \mathbb{I}_{n+1} \qquad \mathsf{id}\, x = x$$

$$\_\,,\_ : X \to (\mathbb{I}_n \to X) \to (\mathbb{I}_{n+1} \to X)$$
$$(s\,,\sigma)\ 0 = s$$
$$(s\,,\sigma)\ (\uparrow x) = \sigma\, x$$

$$\_ \circ \_ : (B \to C) \to (A \to B) \to (A \to C)$$
$$(\sigma \circ \tau)\, x = \sigma(\tau x)$$

Figure 1. Primitives of the σ-calculus.

k-element type $\mathbb{I}_k$. This corresponds to a reference in a context of size k.

Syntactic sorts are then represented by $\mathbb{N}$-indexed inductive families. The scopes are exclusive upper bounds on the freely occurring variables of certain sorts.

The following grammar gives System F types in a well-sorted de Bruijn representation:

$$A^n, B^n \in ty_n \ ::= \ x^n_{ty} \mid A^n \to B^n \mid \forall.\ A^{n+1} \qquad x \in \mathbb{I}_n$$

Note how in the case of universal quantification the type index of A increases by 1 for the additional freely occurring variable. Although all types are scoped, we usually leave out the indices for the sake of readability.

Instantiation. We recall the definition of instantiating a type A with a parallel type substitution $\sigma : \mathbb{I}_m \to ty_n$, written $A[\sigma]$.

For the definition, we use a well-scoped version of the primitive operations first defined in the σ-calculus [Abadi et al. 1991], as depicted in Figure 1. E.g., the *stream cons* substitution $A\,,\sigma$ maps the index 0 to A and indices $\uparrow x$ to $\sigma\, x$. Note that this operation binds weaker than composition.

A substitution acts on all k free type variable in A^k at once. We define $A[\sigma]$ mutually recursive with the forward composition of substitutions:

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

Substitution traverses the term homomorphically. If we reach the variable constructor, we perform the substitution. Traversing a universal quantifier changes the interpretation of indices in scope. We thus adjust the substitution via a *lifting operation* $\Uparrow^{ty}$. The index 0 is mapped to 0, indices of the form $\uparrow x$ are first mapped to $\sigma\, x$ and then adjusted to bypass the new binder. We achieve this adjustment by simply post-composing $\uparrow$ to σ.

The σ-calculus. Single-point substitutions, i.e., substitutions which only act on *one* variable, interfere with each

$$(s\,,\sigma)\, 0 = s \qquad (s\,,\sigma) \circ \uparrow\ \equiv \sigma$$
$$(s\,,\sigma)\, (\uparrow n) = \sigma\, n \qquad \sigma\, 0\,, \uparrow \circ\, \sigma \equiv \sigma$$
$$0\,, \uparrow\ \equiv \mathsf{id} \qquad (s\,,\sigma) \circ \tau \equiv \tau\, s\,, \sigma \circ \tau$$
$$\sigma \circ \mathsf{id} \equiv \sigma \qquad \mathsf{id} \circ \sigma \equiv \sigma$$

(a) Interference laws.

$$X[\sigma] = \sigma\, X$$
$$(A \to B)[\sigma] = A[\sigma] \to B[\sigma]$$
$$(\forall.\ A)[\sigma] = \forall.\ A[\Uparrow^{ty} \sigma] \qquad \text{with } \Uparrow^{ty} \sigma = 0_{ty}\,, \sigma \circ [\uparrow]$$

(b) Reduction laws.

$$\mathsf{id}_{ty} \circ \_[\sigma] \equiv \sigma$$
$$s[\mathsf{id}_{ty}] = s$$
$$s[\sigma][\tau] = s[\sigma \circ [\tau]]$$

(c) Monad laws.

$$\_[\mathsf{id}_{ty}] \equiv \mathsf{id}$$
$$(\sigma \circ [\tau]) \circ [\theta] \equiv \sigma \circ [\tau \circ [\theta]]$$

(d) Supplementing laws.

Figure 2. Rewriting system for System F types.

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.

This parallel-substitution equational theory was first presented in [Abadi et al. 1991] for the λ-calculus and consists of four categories of laws (Figure 2).

Interference laws govern the interplay of basic forward composition, stream cons, and shifting, while *reduction laws* account for the sort-specific definition of instantiation. Instantiation on terms can be seen as a monad from a category of renamings to the terms themselves [Altenkirch et al. 2010] and thus satisfies the *monad laws*, i.e. left identity, right identity, and composition. Last, the above rules might generate critical pairs. The *supplementing laws* ensure that the rewriting system is confluent.

Note that several equations are stated as equivalences, where $f \equiv g$ if $f\, x = g\, x$ for all arguments x.

The resulting equational theory is terminating [Abadi et al. 1991], confluent [Curien et al. 1996], and complete [Schäfer et al. 2015a].

Autosubst 1. Autosubst 1 simplifies reasoning with parallel de Bruijn substitutions for generalised syntax. As such, it takes an annotated inductive type of terms, and generates a model for a sort-specific version of the σ-calculus described above using Ltac. This includes both the derivation of the capture-avoiding instantiation operation for parallel substitutions and the corresponding lemmas.

The implementation requires several intermediate steps. First, note that the mutual recursion between instantiation and composition is not structural. We follow the pattern presented in [Adams 2004] and first define instantiation for the special case of (not necessarily injective) renamings, i.e. substitutions which only substitute variables.

Next, the lemmas of the aforementioned equational theory have to be proven. While the interference laws hold independent of a specific syntax, the reduction laws and left identity follow immediately from our definition of instantiation itself. In contrast, the two remaining monad laws require several inductions (e.g., compositionality requiring first instances where either σ, or τ, or both σ and τ are renamings). Last, the supplementing laws follow with the corresponding monad laws.

Coq provides a type theory *without* functional extensionality, i.e. $f \equiv g \nrightarrow f = g$. However, as functional extensionality may be safely assumed [Hofmann 1995], which Autosubst does to provide a tactic `asimpl` that automatically rewrites the lemmas in Figure 2.

Specifications. Autosubst 2 takes a second-order HOAS specification as input. A specifications is a context Θ declaring new types and constructors.

$$\begin{array}{rrcl} \text{(specifications)} & \Theta & ::= & T_1 : \text{Type}, \dots, C_1 : U_1, \dots \\ \text{(constructor types)} & U & ::= & T \mid (T_1 \to \dots \to T_n) \to U \end{array}$$

Since we only allow simple types, our specifications are fairly restrictive compared to contemporary type theories supporting HOAS [Pfenning and Schürmann 1999; Pientka and Dunfield 2010]. In fact, our specifications are equivalent to the multi-sorted second-order binding signatures of Ahrens and Zsido [2011].

We chose the HOAS presentation to allow us to extend Autosubst 2 in the future. Tools like Twelf and Beluga have already shown that it is possible to include type systems (by adding dependent types and a universe of propositions) and recursive definitions (e.g., by adding modalities [Hofmann 1999a; Nanevski et al. 2008]) in the same formalism.

3 From Parallel Substitutions to Vector Substitutions on the Example of F_{CBV}

In Section 2 we saw the instantiation of renaming and substitution for *one* sort of variables. Let us consider an example with multiple sorts of variables: a call-by-value variant of System F.

$$\begin{array}{rcll} A^k, B^k \in ty^k & ::= & x^k_{ty} \mid A^k \to B^k \mid \forall.\ A^{k+1} & x \in \mathbb{I}_k \\ s^{k,l}, t^{k,l} \in tm^{k,l} & ::= & s^{k,l}\ t^{k,l} \mid s^{k,l}\ A^k \mid v^{k,l} & \\ u^{k,l}, v^{k,l} \in vl^{k,l} & ::= & x^{k,l}_{vl} \mid \lambda A^k.\ s^{k,l+1} \mid \Lambda.\ s^{k+1,l} & x \in \mathbb{I}_l \end{array}$$

While its sort of types is still univariate, i.e. contains only one type of variables, the terms s and values v are multivariate, i.e. both type and value variables can be bound. Terms and values are thus indexed by the upper bound of both type and value variables. F_{CBV} concisely showcases the complications that arise from a potentially mutual syntax and we thus use it as our example syntactic system in the remainder of the section.

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

$$\begin{array}{ll} \Uparrow^{vl}\ \sigma := \sigma & \Uparrow^{vl}\ \tau := 0_{vl}\,, \tau \circ [\mathrm{id}_{ty}; \uparrow] \\ \Uparrow^{ty}\ \sigma := 0_{ty}\,, \sigma \circ [\uparrow] & \Uparrow^{ty}\ \tau := \tau \circ [\uparrow; \mathrm{id}_{vl}] \end{array}$$

Figure 3. Term and value substitutions for F_{CBV}.

Extending instantiation to terms and values implies the need to substitute for both type and value variables. In Autosubst 1, this resulted in different instantiation operations for type and value variables, e.g. for values:

$$\_[\_]_{ty} : vl_{m\,n} \to (\mathbb{I}_m \to ty_{m'}) \to vl_{m'\,n}$$
$$\_[\_]_{vl} : vl_{m\,n} \to (\mathbb{I}_n \to vl_{m\,n'}) \to vl_{m\,n'}$$

This results in a total of five instantiation operations.

We face the problem that the various instantiation operations interfere and become difficult to permute. Take for example

$$s[\tau]_{vl}[\sigma]_{ty} = s[\sigma]_{ty}[\sigma \circ [\tau]_{ty}]_{vl}$$

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

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 parallel substitutions, with one component for each sort that may occur in a variable position. We will see that this leads to a more uniform treatment and a simple equational theory. In the following, we give the required definitions for F_{CBV} to illustrate the approach.

Though we use F_{CBV} as a running example, note that the Autosubst 2 tool extends the above approach to all second-order HOAS signatures.

3.1 Instantiation

See Figure 3 for a definition of instantiation

$$\_[\_;\_] : vl_{m\,n} \to (\mathbb{I}_m \to ty_{m'}) \to (\mathbb{I}_n \to ty_{m'\,n'}) \to vl_{m'\,n'}$$

for well-scoped System F.

The instantiation operations for terms and values are defined in Figure 3, 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 $[\sigma]$. As the shape of the substitution will be determined from a *transitive* notion of occurrence, a suitable subvector will always exist.

Third, and most interestingly, the traversal of binders changes the interpretation of the indices in scope. We have to adjust each component of the substitution vector via a customised *lifting operation* which is more involved than in the single-sorted setting (cf. the overloaded $\Uparrow^{ty}$). 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 (capture-avoiding substitution should not substitute the newly bound variable), we have to ensure that $\uparrow x$ is first mapped to σx and then adjusted to bypass the new binder. For types, this was achieved by simply post-composing $\uparrow$ to σ. Instead, we have to post-compose a vector substitution which matches the codomain of σ, has a shift for the bound sort, and is otherwise the variable constructor $\mathsf{id}_{ty} : \forall n.\, \mathbb{I}_n \to \mathbb{I}_n$ – corresponding to the change of interpretation under the corresponding binder. We further have to construct and post-compose such adjustments to all other components σ' of the original vector substitution. For our concrete example, these are the two operations $\Uparrow^{vl}$ and $\Uparrow^{ty}$ defined in Figure 3 both of which construct substitutions suitable for the sort of terms while incorporating a newly bound value or, respectively, type. When we observe their uses, we see that they act on substitutions for the sort of values, indicating that the subvector cast mentioned above may, in fact, be the identity. We further observe that the post-composed 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.

Coq Implementation. Again, the mutual recursion between instantiation and composition is not structural, and we thus first define instantiation for renamings, written $A\langle\xi\rangle$ and $s\langle\xi;\zeta\rangle$. For example, $\Uparrow^{ty}$ is in fact defined as

$$\Uparrow^{ty} \sigma := 0_{ty}\,, \sigma \circ \langle\uparrow\rangle$$

and similarly for $\Uparrow^{vl}$.

3.2 Rewriting System

Based on the aforementioned aspects we extend the σ-calculus [Abadi et al. 1991] to vector substitutions. We recall the different kinds of rules of the σ-calculus (Figure 2): interference, reduction, monad, and supplementing laws. As no new primitives were needed, the interference laws remain unchanged. As before, the reduction laws have to be adapted

Laws of System F types
+ reduction laws
+

$$\begin{aligned}
s[\mathsf{id}_{ty};\mathsf{id}_{vl}] &= s\\
s[\sigma_{ty};\sigma_{vl}][\tau_{ty};\tau_{vl}] &= s[\sigma_{ty}\circ[\tau_{ty}];\sigma_{vl}\circ[\tau_{ty};\tau_{vl}]]\\
\mathsf{id}_{vl}\circ _[\sigma;\tau] &\equiv \tau\\
v[\mathsf{id}_{ty};\mathsf{id}_{vl}] &= v\\
v[\sigma_{ty};\sigma_{vl}][\tau_{ty};\tau_{vl}] &= v[\sigma_{ty}\circ[\tau_{ty}];\sigma_{vl}\circ[\tau_{ty};\tau_{vl}]]
\end{aligned}$$

(a) Monad laws.

$$\begin{aligned}
_[\mathsf{id}_{ty};\mathsf{id}_{vl}] &\equiv \mathsf{id}\\
(\sigma_{vl}\circ[\tau_{ty};\tau_{vl}])\circ[\theta_{ty};\theta_{vl}] &\equiv \sigma_{vl}\circ[\tau_{ty}\circ[\theta_{ty}];\tau_{vl}\circ[\theta_{ty};\theta_{vl}]]
\end{aligned}$$

(b) Supplementing laws.

Figure 4. Equational System for Terms and Values of System F.

to the extended syntax. The monad laws and supplementing laws further hold in a generalised form, we account for in the following.

In the case of right identity, the single identity substitution extends to a vector of identity substitutions. We prove the statement as follows:

Lemma 3.1 (Right Identity). *$s[\mathsf{id}_{ty},\mathsf{id}_{vl}] = s$ and $v[\mathsf{id}_{ty},\mathsf{id}_{vl}] = v$.*

Proof. By a mutual induction on s and v. In the case of the value variable constructor, the goal holds directly by definition of id_{vl}.

Otherwise, in each case, the statement holds by congruence and the corresponding proof on the single components. E.g., in the case of application, $s\,A$, we show that $s[\mathsf{id}_{ty},\mathsf{id}_{vl}] = s$ and $A[\mathsf{id}_{ty}] = A$. Note that for A, the statement relies on the respective proofs for ty.

Next, in each case we traverse a binder, we have to account for the scope change. For example, in the case of type abstraction, we have to show that $s[\Uparrow^{ty}\mathsf{id}_{ty},\Uparrow^{ty}\mathsf{id}_{vl}] = s$. This requires two additional proofs to show that $\Uparrow^{ty}\mathsf{id}_{ty}\equiv\mathsf{id}_{ty}$ and $\Uparrow^{ty}\mathsf{id}_{vl}\equiv\mathsf{id}_{vl}$.

If the lifted variable does *not* correspond to the newly bound variable (e.g., for $\Uparrow^{ty}\mathsf{id}_{vl}$), this follows directly. Otherwise, e.g. for $\Uparrow^{ty}\mathsf{id}_{ty}$, we have to take a case analysis on the examined variable, where again in both cases the statement follows directly. □

Note how this proof followed the structure of instantiation: We had to take care of the mutual structure, the correct subcomponents, the variable constructor, and adequate lifting operations.

We prove compositionality in a similar manner. As in Autosubst 1, we have to show the statement for all combinations of renamings and substitutions (cf. [Adams 2004]):

Lemma 3.2 (Compositionality)**.**

1. $s\langle\xi_{ty};\xi_{vl}\rangle\langle\zeta_{ty};\zeta_{vl}\rangle = s\langle\xi_{ty} \circ \zeta_{ty};\xi_{vl} \circ \zeta_{vl}\rangle$
2. $s\langle\xi_{ty};\xi_{vl}\rangle[\tau_{ty};\tau_{vl}] = s[\xi_{ty} \circ \tau_{ty};\xi_{vl} \circ \tau_{vl}]$
3. $s[\sigma_{ty};\sigma_{vl}]\langle\zeta_{ty};\zeta_{vl}\rangle = s[\sigma_{ty} \circ \langle\zeta_{ty}\rangle;\sigma_{vl} \circ \langle\zeta_{ty};\zeta_{vl}\rangle]$
4. $s[\sigma_{ty};\sigma_{vl}][\tau_{ty};\tau_{vl}] = s[\sigma_{ty} \circ [\tau_{ty}];\sigma_{vl} \circ [\tau_{ty};\tau_{vl}]]$

Proof. Again, each statement will require an induction on s, following the above traversal structure. Each scope change will require corresponding lifting lemmas, which will, in turn, require the previous compositionality statements. We omit the details of the proofs. ◻

Finally, we have to adapt the supplementing laws.

Lemma 3.3 (Supplementing Laws)**.**

1. $\_[\mathsf{id}_{ty};\mathsf{id}_{vl}] \equiv \mathsf{id}$
2. $(\sigma_{vl} \circ [\tau_{ty};\tau_{vl}]) \circ [\theta_{ty};\theta_{vl}]$
   $\equiv \sigma_{vl} \circ [\tau_{ty} \circ [\theta_{ty}];\tau_{vl} \circ [\theta_{ty};\theta_{vl}]]$

Proof. Both statements follow with right identity respectively compositionality. ◻

We conjecture that the extended term rewriting system is still confluent and terminating.

Note that in the previous statement we silently assumed extensionality. For the monad law, we can avoid extensionality with more general statements (see Section 5). Otherwise, the following law will provide useful:

Lemma 3.4 (Extensionality)**.**

$$\frac{\sigma_{ty} \equiv \tau_{ty} \qquad \sigma_{vl} \equiv \tau_{vl}}{s[\sigma_{ty};\sigma_{vl}] = s[\tau_{ty};\tau_{vl}]}$$

Proof. By induction on s, requiring first an instance for renaming. The traversal structure is similar to the right identity law. ◻

3.3 Typing and Evaluation for $\mathrm{F_{CBV}}$

Let us examine how we can use vector substitutions to state evaluation and typing for $\mathrm{F_{CBV}}$ (Figure 5).

For evaluation, vector substitutions allow us to state β-reduction for both types of abstraction. In the case of term abstraction, we lift the term component and the type component remains unchanged, vice versa for type abstraction.

As we work in a well-scoped syntax, term typing is stated as a predicate of type

$$\forall m\, n.(\mathbb{I}_m \to ty_n) \to tm_{m;n} \to ty_n \to \mathbb{P},$$

i.e. contexts are represented as functions. The variable rule will thus simply state that a variable has the type its context dictates: $\Gamma \vdash^v x : \Gamma\, x$. In the case of type abstraction, the scope changes, which we come up for by composition of the necessary shifting operation.

In the next section, we use our rewriting system to prove preservation.

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

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

$$\frac{\Gamma \vdash^v v : A}{\Gamma \vdash v : A}$$

$$\Gamma \vdash^v x : \Gamma\, x \qquad \frac{\Gamma, A \vdash s : B}{\Gamma \vdash^v \lambda A.\, s : A \to B} \qquad \frac{\Gamma \circ \langle\uparrow\rangle \vdash s : A}{\Gamma \vdash^v \Lambda.\, s : \forall.\, A}$$

Figure 5. Evaluation and type system of $\mathrm{F_{CBV}}$.

4 First-Class Renamings

In Autosubst 1 and the σ-calculus, renamings are treated as a special case of substitutions, and are thus second class. While convenient for the equational theory, this does not match a proof assistant which supports only structural induction:

Often, substitution properties might require proving the corresponding instance for renamings first. A well-known example are context morphism lemmas [Goguen and McKinna 1997; Kaiser et al. 2017b], stating e.g. that typing of $\mathrm{F_{CBV}}$ (Figure 5) is substitutive:

$$\frac{\Gamma \vdash s : A \qquad \forall x. \Delta \vdash \tau\, x : (\Gamma\, x)[\sigma]}{\Delta \vdash s[\sigma;\tau] : A[\sigma]}$$

The proof proceeds by induction on $\Gamma \vdash s : A$. In the case of abstraction the context changes to Γ, A and we need to show that the precondition is preserved under it, i.e.,

$$A\,, \Delta \vdash (\Uparrow^{tm} \tau)\, x : ((A\,, \Gamma)\, x)[\Uparrow^{tm} \sigma]$$

for all x. For an arbitrary x, this statement is no longer covered by the inductive hypothesis and we will thus first require a proof for renamings of the form

$$\frac{\Gamma \vdash s : A \qquad \forall x. (\Gamma\, x)\langle\xi\rangle = \Delta\,(\zeta\, x)}{\Delta \vdash s\langle\xi;\zeta\rangle : A\langle\xi\rangle}$$

Even worse: Some statements only hold for renaming, such as the anti-renaming lemma,

$$\text{injective } \xi \to s\langle\xi\rangle \Downarrow t\langle\xi\rangle \to s \Downarrow t.$$

In Autosubst 1, we can only reason about a renaming ξ by an embedding into substitutions via $[\xi \circ \mathsf{id}_{ty}]$. This complicates the application of previous inductive hypotheses and makes working with renamings unnecessarily cumbersome.

In Autosubst 2, we decided to make renamings first-class, similar to [Pitts 2013]. For example, in the case of System F types, there are both renamings $\xi : \mathbb{I}_m \to \mathbb{I}_n$ and substitutions $\sigma : \mathbb{I}_m \to tm_n$. At the same time, we allow the user to employ both instantiation of renamings ($\xi\langle s\rangle$) and instantiation of substitutions ($[s]\sigma$).

The resulting rewriting system will thus hold variants for both instantiation of renamings and substitutions. For example, in the case of the right identity law, the law will be stated as:

$$s\langle \mathsf{id}; \mathsf{id}\rangle = s.$$

Compositionality will require four instances, one for each combination of renaming and substitution. Note that in the *definition* of instantiation and in the proofs for the equational theory, we did first use renamings to resolve the mutual inductive structure. So all the infrastructure already exists.

Renamings are always preserved unless the user explicitly decides to do otherwise (see Section 5.5). However, we provide the following law connecting instantiation of renamings and substitutions:

Lemma 4.1 (Renaming and Instantiation)**.**

$$s[\xi \circ \mathsf{id}_{ty}] = s\langle \xi \rangle$$

Proof. By induction on s. The proof extends to vector substitutions and follows the expected traversal structure. □

The resulting equational theory of a σ-calculus with renamings still has to be examined in detail. We conjecture that convergence and termination are preserved, even though completeness breaks.

Let us revisit substitutivity of typing and see how Autosubst performs.

Lemma 4.2 (Context Renaming Lemma)**.**

$$\frac{\Gamma \vdash s : A \qquad \forall x.(\Gamma\, x)\langle \xi \rangle = \Delta\,(\zeta\, x)}{\Delta \vdash s\langle \xi; \zeta \rangle : A\langle \xi \rangle}$$

and

$$\frac{\Gamma \vdash s : A \qquad \forall x.\Delta \vdash \tau\, x : (\Gamma\, x)[\sigma]}{\Delta \vdash s[\sigma; \tau] : A[\sigma]}$$

Proof. By induction on $\Gamma \vdash s : A$. All cases without a change of scope follow immediately.

For context renaming and term abstraction, we have to show that

$$\Delta \vdash \lambda A[\xi].\ s[\Uparrow^{tm}\ \xi; \Uparrow^{tm}\ \zeta] : (A \to B)[\xi]$$

and thus with the inductive hypothesis that

$$\forall x.((A\,,\Gamma)\,x)\langle \Uparrow^{tm}\ \xi \rangle = A\langle \xi \rangle\,,\Delta\,((\Uparrow^{tm}\ \zeta)\,x)$$

This requires a case analysis on x. In both cases, the statement follows with the primitive equations for stream cons.

The same holds for the context substitution lemma, but this time we have to show that

$$A\,,\Delta \vdash (\Uparrow^{tm}\ \tau)\,x : ((A\,,\Gamma)\,x)[\Uparrow^{tm}\ \sigma]$$

This time, we will need the variable case of typing for $x = 0$. In the case of $\uparrow x$, we have to apply the context renaming lemma and need to use right identity to show that $(\Gamma x)[\sigma] = (\Gamma x)[\sigma]\langle \mathsf{id}\rangle$. To show the premise of the context renaming lemma, we need again the definition of stream cons and right identity.

All of the above equations can be solved automatically using the Autosubst 2 framework. □

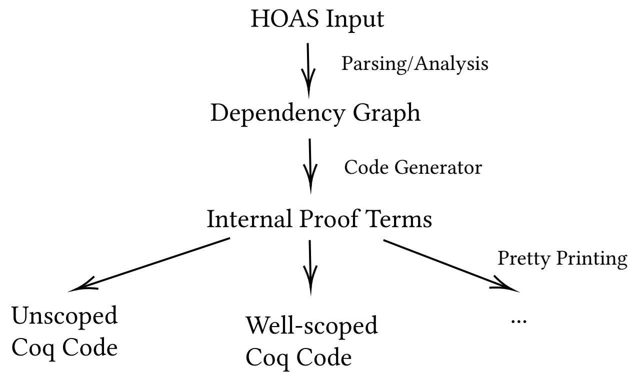

Figure 6. Set-up of Autosubst 2.

5 From HOAS to Vector Substitutions

All definitions and statements of Section 3 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 implementation in Haskell parses a Twelf-like second order HOAS system specification and produces the desired output as a plain source file which can be read by the proof assistant. See Figure 6 for an overview of the set-up. From the HOAS system specification, we deduce which sorts require variables, which dependencies exist, and the minimal vectors needed for instantiation. This is similar to the subordination analysis in Beluga [Pientka and Dunfield 2010] or Twelf [Pfenning and Schürmann 1999].

Based on this information, we construct the required model of the σ-calculus in an intermediate type theory syntax, which abstracts from the specific proof assistant. In the last step, we use pretty printing to generate the required proof script.

In the following, we explain the subcomponents of Autosubst 2 in more detail.

5.1 Generation of the Dependency Graph

A sample input specification and the desired well-scoped inductive term sorts for F_{CBV} are shown in Figure 7.

To understand why a HOAS specification suffices 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,
```

```
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 n : Type :=
| var_ty : fin n → ty n
| arr : ty n → ty n → ty n
| all : ty (n + 1) → ty n.

Inductive tm m n: Type :=
| app : tm m n → tm m n→ tm m n
| tapp : tm m n → ty m → tm m n
| vt : vl m n → tm m n
with vl m n : Type :=
| var_vl : fin n → vl m n
| lam : ty n → tm m (n + 1) → vl m n
| tlam : tm (m + 1) n → vl m n.
```

Figure 7. HOAS specification of F_{CBV} (left) and the corresponding inductively defined, well-scoped de Bruijn sorts (right).

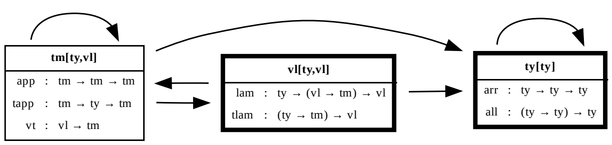

Figure 8. Dependency graph of F_{CBV}.

we will refer to the result type of each argument as the *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 occurrence 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.

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 F_{CBV} is shown in Figure 8. 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`.

5.2 Generation of Internal Proof Terms

The dependency graph yields all the information needed to define a model of the σ-calculus for the corresponding term sort. We process this dependency graph in topological order, preserving the input order of sorts and constructors as much as possible, to generate the desired output.

The following aspects will be relevant (see Section 3.2): First, the definitions and proofs follow the inductive structure of the term sorts. 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. In case the respective sort contains a variable constructor, we have to handle this case separately. Projections have to be respected. We use no explicit projections, but handle these by omitting irrelevant invariants. New binding constructs will require special treatment: Either in the form of a changed definition or in the case of the lemmas applying a previously defined lemma which states that the invariants are preserved.

In the following we discuss the generation of the different components.

Inductive term sorts. 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. Sorts in the same strongly connected component have to be declared mutually inductive. If a term is bound, the index of the respective term is increased by 1.

Instantiation. Instantiations are slightly more interesting. As we work in a proof assistant with only structural recursion, we declare instantiation first for renamings, then for substitutions. Both are defined by a mutual recursion on all inter-dependent sorts. Recall Section 3.1 for a definition. For the correct choice of a lifting 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 lifting operations have to be generated, and how.

Lemmas. In Section 3, we showed how to prove the required substitution lemmas.

In contrast to Autosubst 1, we state the lemmas in a more general fashion which does not require functional extensionality. For example, compositionality for the terms and values of F_{CBV} is stated as:

$$\frac{\theta_{ty} \equiv \sigma_{ty} \circ [\tau_{ty}] \qquad \theta_{vl} \equiv \sigma_{vl} \circ [\tau_{ty}; \tau_{vl}]}{s[\sigma_{ty}; \sigma_{vl}][\tau_{ty}; \tau_{vl}] = s[\theta_{ty}; \theta_{vl}]}$$

The proof still follows the previously defined structure.

Technical Realisation. In the generation of proof terms, it is crucial to be as precise as possible: This prevents generalisation problems or problems specific due to proof-assistant-specific behaviour. This includes that we (1) use as few implicit arguments as possible, (2) do not use any notation during definition, and (3) use proof terms instead of tactics. The last point further ensures that we can extend Autosubst 2 to proof assistants without a tactic interface, e.g. Agda.

To simplify generation, we introduced a special type for substitution objects. This type includes pre-defined instances of scope vectors, renaming vectors, substitution vectors, and the possibility to include equation vectors, needed to prove the monad lemmas. Together with this data type we define specific functions to select subvectors or to lift the object into a new scope, e.g. for renamings by post-composing the corresponding vector.

Moreover, we implemented a traversal-like function to simplify the definition of both instantiation and the lemmas for the rewriting system. The function basically implements traversals as described in [Allais et al. 2017; Kaiser et al. 2018], traversing a term and changing the scope if necessary.

5.3 Printing

In the last step, we transform the internal proof terms into plain text, readable (and checkable) by a proof assistant of choice. We use pretty printing [Wadler 2003] together with Haskell's type class mechanism. Every backend comes with its own type class for printing, each internal construct requires a corresponding instance of the type class.

Treating substitution objects special proved useful: The well-scoped and unscoped variant of the generated Coq code only differ in the `Show` instance of scoping objects.

Adapting Autosubst 2 to a new proof assistant (with the corresponding support for, e.g., mutually inductive types) is easy. Printing for Lean [de Moura et al. 2015] and Agda [Norell 2008] are work-in-progress.

5.4 Automation

Every instance of automation should implement the rewriting system of the σ-calculus.

Under the assumption of functional extensionality, we can simply rewrite the corresponding lemmas of the rewriting system. The current implementation does exactly this in few lines of Ltac. The tactic `asimpl` rewrites the corresponding lemmas in the goal, respectively for `asimpl in *` in both all hypotheses and in the goal. The reduction laws are not rewritten, but directly reduced using Coq's evaluation tactic, to reduce proof term size. Notations (Section 5.5) have to be unfolded.

A tactic without functional extensionality is possible with our restricted syntax, but has to traverse the term. This requires the (already automatically generated) extensionality lemma (Lemma 3.4). An implementation is currently work-in-progress.

```
(* Type Class for the Notation *)
Class Subst1 (X1 : Type) (Y Z: Type) :=
  subst1 : X1 → Y → Z.

Notation "s [ sigma ]" := (subst1 sigma s) (...).

(* Declared instance of the notation. *)
Instance Subst_ty { mty nty : nat } :
Subst1 (fin (mty) → ty (nty)) (ty (mty)) (ty (nty))
:= subst_ty (mty) (nty) .

(* Additional Printing Notation. *)
Notation "s [ σ ]" := (subst_ty σ s) (..., only printing).
```

Figure 9. Instantiation notation F_{CBV} types.

5.5 Notation

We aim for a univariate syntax for instantiation for different sorts, i.e. the user should be able to write $s[\sigma]$ without knowing the exact name of the specific instantiation operation.

We use a common type class instance to be able to overload the parsing of notation (see Figure 9) in Coq. Autosubst 2 generates the required instances together with the remaining code. Each instance is unique because of its result type. As automation works on terms without notation, `asimpl` will need to unfold the type class instances.

Folding (dependent) instances is difficult and we thus introduce a all notations a second time just for printing (Figure 9). As such, we will never fold *to* a type class instance.

We have introduced similar syntax for renamings ($A\langle\xi\rangle$ and $s\langle\xi;\zeta\rangle$), variable constructors (`ids`), and the lifting of variables. For example, a scope change that lifts one type variable van be written as $\Uparrow^{ty}\ \sigma$, independent of the underlying type of σ.

Renamings and Substitutions. At the moment, our automation tactic does not automatically switch between renamings and substitutions.

Instead, we provide tactics which allow the user to automatically transform renamings to substitutions (`substify`) and vice versa (`renamify`).

The first direction is the easy one, simply rewriting with Lemma 4.1 from left to right. On the other side, `renamify` requires several transformations: A substitution of the form $\xi \circ \mathsf{id}_{ty}$ can be directly transformed to a renaming, otherwise, we have to re-parenthesise to the left and then re-try. Moreover, we might have to fold the stream cons, e.g. from $\mathsf{id}_{ty} x\ ,(\xi \circ \mathsf{id}_{ty})$ to $(x\ , \xi) \circ \mathsf{id}_{ty}$.

6 Case Studies

We developed several case studies to test the performance of Autosubst 2. All developments can be found online. See the context renaming lemma in Section 4 for a detailed proof using the rewriting system of Autosubst 2.

POPLMark Reloaded. We provide a solution for the POPLMark Reloaded challenge [Abel et al. 2017], in which we show strong normalisation for a strong λ-calculus with disjoint sums using Kripke logical relations for both unscoped and well-scoped syntax. A well-scoped solution, which provides for elegant handling of contexts, would not have been possible with Autosubst 1.

Algorithmic Equivalence. We revisit a proof of the equivalence of algorithmic and definitional equivalence for an untyped λ-calculus via logical relations [Crary 2005]. We follow the structure of the corresponding Beluga proof [Cave and Pientka 2015], and, similarly, omit units. The proof uses only one syntactic sort but relies on the convenient handling of renamings.

POPLMark Challenge (Part A). In the next section, we present a proof of part A of the POPLMark challenge using vector substitutions. This simplifies on the previous Autosubst 1 proof with ad-hoc heterogeneous substitutions.

Weak Normalisation of F_{CBV}. We present a concise formal proof that F_{CBV} is weakly normalising using a unary logical relation. The logical relation interprets (open) types as mappings from environments to sets of closed values. 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. The proof requires two sorts of variables and mutual inductive syntax.

Call-by-push-value. We use Autosubst 2 in a recent 8000-line development [Forster et al. 2018] formalising the operational, equational, and denotational semantics of call-by-push-value [Levy 1999], a language subsuming the call-by-value/call-by-name λ-calculus with sums and products. Autosubst 2 excels in a development with three different sorts with variables, two of them (fine-grained CBV and CBPV) mutually inductive. Mutually inductive types and well-scoped syntax were not supported in Autosubst 1, and thus this case study would not have been possible.

6.1 Weak Normalisation of F_{CBV}

We present a concise formal proof that F_{CBV} is weakly normalising. In Figure 5, we defined both the typing rules and a big-step reduction relation from terms to values.

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 $d\,,\rho$ 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 $[\![A]\!]_\rho$ and a value interpretation $(\!|A|\!)_\rho$.

$$\begin{aligned}
[\![A]\!]_\rho &:= \lambda s.\, \exists v.\, s \Downarrow v \wedge (\!|A|\!)_\rho\, v \\
(\!|X|\!)_\rho &:= \rho\, X \\
(\!|A \to B|\!)_\rho &:= \{\, \lambda C.\ s \mid \forall v.\, (\!|A|\!)_\rho\, v \to [\![B]\!]_\rho\, s[\mathrm{id}_{ty}; v\,, \mathrm{id}_{vl}]\,\} \\
(\!|\forall.\ A|\!)_\rho &:= \{\, \Lambda.\ s \mid \forall Bd.\, [\![A]\!]_{d\,,\,\rho}\, s[B\,, \mathrm{id}_{ty}, \mathrm{id}_{vl}]\,\}
\end{aligned}$$

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

Lemma 6.1. *For all types A, environments ρ, and renamings ξ we have $(\!|A[\xi]|\!)_\rho = (\!|A|\!)_{\xi\,\circ[\rho]}$. In particular, $(\!|A[\uparrow]|\!)_{d\,,\,\rho} = (\!|A|\!)_\rho$ holds.*

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

Lemma 6.2. *For all types A, environments ρ, and substitutions σ we have $(\!|A[\sigma]|\!)_\rho = (\!|A|\!)_{\sigma\,\circ(\!|-|\!)_\rho}$. The result trivially lifts to the term interpretation and we obtain $[\![A[B\,, \mathrm{id}_{ty}]]\!]_\rho = [\![A]\!]_{(\!|B|\!)_\rho\,,\,\rho}$ as a special case.*

Proof. Induction on A using Lemma 6.1. □

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

$$\begin{aligned}
(\!|\Gamma|\!)_\rho &:= \lambda\tau.\, \forall x.\, (\!|\Gamma_x|\!)_\rho\, (\tau\, x) \\
\Gamma \vDash s : A &:= \forall \sigma\tau\rho.\, (\!|\Gamma|\!)_\rho\, \tau \to [\![A]\!]_\rho\, s[\sigma;\tau] \\
\Gamma \vDash^v v : A &:= \forall \sigma\tau\rho.\, (\!|\Gamma|\!)_\rho\, \tau \to (\!|A|\!)_\rho\, v[\sigma;\tau]
\end{aligned}$$

We now prove that syntactic typing implies semantic typing.

Theorem 6.3 (Soundness)**.** *For all Γ, s, v, A we have*

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

Proof. By mutual induction on the typing derivations. The type application case introduces a substitution on types which is handled with Lemma 6.2. Meanwhile, type abstraction relies on Lemma 6.1. The proof also depends on two non-trivial substitution lemmas for the cases of abstraction and type abstraction.

$$\begin{gathered}
s[\Uparrow^{vl}\ (\sigma;\tau)][\mathrm{id}_{ty}; v\,, \mathrm{id}_{vl}] = s[\sigma; v\,, \tau] \\
s[\Uparrow^{ty}\ (\sigma;\tau)][A\,, \mathrm{id}_{ty}; \mathrm{id}_{vl}] = s[A\,, \sigma;\tau]
\end{gathered}$$

Both are solved automatically by our framework. □

Corollary 6.4 (Weak Normalisation)**.** *For all s, A we have*

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

Syntax of $F_<$

$$A^k, B^k \in ty^k \ ::= \ x^k_{ty} \mid A^k \to B^k \mid \forall <: A^k.B^{k+1} \qquad x \in \mathbb{I}_k$$

$$s^{k,l}, t^{k,l} \in tm^{k,l} \ ::= \ x^{k,l}_{tm} \mid s^{k,l}\, t^{k,l} \mid s^{k,l}\, A^k$$

$$\lambda A^k.\ s^{k,l+1} \mid \Lambda <: .[k,l]As^{k+1,l} \qquad x \in \mathbb{I}_l$$

Subtyping $\Delta \vdash A <: B$

$$\Delta \vdash A <: T \qquad \Delta \vdash x <: x \qquad \frac{\Delta \vdash \Delta\, x <: B}{\Delta \vdash x <: B}$$

$$\frac{\Delta \vdash B_1 <: A_1 \qquad \Delta \vdash A_2 <: B_2}{\Delta \vdash A_1 \to A_2 <: B_1 \to B_2}$$

$$\frac{\Delta \vdash B_1 <: A_1 \qquad (B_1, \Delta) \circ \langle\uparrow\rangle \vdash A_2 <: B_2}{\Delta \vdash \forall <: A_1.A_2 <: \forall <: B_1.B_2}$$

Typing $\Delta; \Gamma \vdash s : A$

$$\frac{\Delta;\Gamma \vdash s : A \to B \qquad \Delta;\Gamma \vdash t : A}{\Delta;\Gamma \vdash s\, t : B} \qquad \frac{\Delta;\Gamma \vdash s : \forall <: A.}{\Delta;\Gamma \vdash s\, B : A[B\,, \mathsf{id}_{ty}]}$$

$$\Delta;\Gamma \vdash x : \Gamma\, x \qquad \frac{\Delta; A, \Gamma \vdash s : B}{\Delta;\Gamma \vdash \lambda A.\ s : A \to B}$$

$$\frac{(A\,, \Delta) \circ \langle\uparrow\rangle; \Gamma \circ \langle\uparrow\rangle \vdash s : B}{\Delta;\Gamma \vdash \Lambda <: A.s : \forall <: A.B} \qquad \frac{\Delta \vdash A <: B \qquad \Delta;\Gamma \vdash s : A}{\Delta;\Gamma \vdash s : B}$$

Weak Semantics $s > t$

$$\frac{s > s'}{s\, t > s'\, t} \qquad \frac{t > t'}{v\, t > v\, t'} \qquad \frac{s > s'}{s\, A > s'\, A}$$

$$\lambda A.\ s\, v > s[\mathsf{id}_{ty}; v\,, \mathsf{id}_{vl}]$$

$$\Lambda <: A.s\, B > s[B\,, \mathsf{id}_{ty}; \mathsf{id}_{vl}]$$

Figure 10. Syntax, typing and weak semantics of $F_<$.

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.

6.2 The POPLMark Challenge (Part A)

We take a second look at part A of the POPLMark challenge [Aydemir et al. 2005]. The goal is to show preservation of $F_<$, i.e. System F with subtyping (Figure 10).

Theorem 6.5 (Preservation)**.** *If* $\Delta;\Gamma \vdash s : A$*, and* $s \Downarrow t$*, then* $\Delta;\Gamma \vdash t : A$.

Typing comes with two kinds of contexts: One for type variables which remembers the subtyping information, one for term variables which collects the typing information. Note that the term context Γ depends on the type context Δ, as its contained objects only make sense in reference to Δ. Thus, a scope change of Δ as for type abstraction requires a scope change on the term context. As contexts are just functions, this can be handled with post-composition.

Our previous solution [Schäfer et al. 2015b] is already based on context morphism lemmas [Goguen and McKinna 1997; Kaiser et al. 2017b]. Unlike [Schäfer et al. 2015b], we use well-scoped syntax with vector substitutions.

In the first part we show that subtyping is transitive and commutes with substitutions. We omit the proofs here, since there were no substantial changes after introducing vector substitutions.

We turn to preservation. We say that $\Delta' <: \Delta$ iff $\Delta' \vdash \Delta'\, x <: \Delta\, x$ for all variables x in scope.

Lemma 6.6. *If* $\Delta' <: \Delta$ *and* $\Delta;\Gamma \vdash s : A$*, then* $\Delta';\Gamma \vdash s : A$.

Proof. By induction on $\Delta;\Gamma \vdash s : A$ with compatibility of subtyping with renaming and substitution. □

Lemma 6.7 (Context Renaming Lemma)**.** *Assume that*

1. $\Delta';\Gamma' \vdash s : A$
2. $\xi\langle(\Delta' x)\rangle = \Delta(\xi x)$ *for all* x
3. $\xi\langle(\Gamma' x)\rangle = \Gamma(\zeta x)$ *for all* x

Then $\Gamma;\Delta \vdash s\langle\xi;\zeta\rangle : A\langle\xi\rangle$.

Proof. By induction on $\Delta';\Gamma' \vdash s : A$, requiring compatibility of subtyping with renaming.

The proof requires equational reasoning with binders in three cases: typing of type application and both term and type abstraction. Let us have a closer look at type abstraction, as this case requires reasoning on both type and value substitutions.

We have to show that $\Delta;\Gamma \vdash \Lambda <: s.\langle\xi;\zeta\rangle : \forall <: A.B$, i.e. $\Delta;\Gamma \vdash \Lambda <: A\langle\xi\rangle.s\langle\Uparrow^{ty}\ \xi; \Uparrow^{ty}\ \zeta\rangle : \forall <: A.B$. Using the corresponding typing rule and the inductive hypothesis, we thus remain with showing that both $A\langle\xi\rangle\,, \Delta \circ \langle\uparrow\rangle$ and $\Gamma \circ \langle\uparrow\rangle$ still fulfill the corresponding preconditions (2) and (3), i.e.

$$\forall x.(\Uparrow^{ty}\ \xi)(((A\,, \Delta') \circ \langle\uparrow\rangle)\, x) = ((A\langle\xi\rangle\,, \Delta) \circ \langle\uparrow\rangle)((\Uparrow^{ty}\ \xi)\, x)$$

and

$$\forall x.(\Uparrow^{ty}\ \xi)\langle(\Gamma' \circ \langle\uparrow\rangle)\, x\rangle = (\Gamma \circ \langle\uparrow\rangle)((\Uparrow^{ty}\ \zeta)\, x),$$

requiring reasoning on the composition of renamings and substitutions, the interaction between cons and composition, and in the first case a case analysis on x.

For both equations, we can use `asimpl` to simplify the goals for us and then use the corresponding property for ξ and Δ to solve the equations. □

Note how using well-scoped syntax allows us to treat contexts as functions themselves, and thus permits the same equational reasoning.

Similarly, we can state the context substitution lemma:

Lemma 6.8 (Context Substitution Lemma). *Assume that*

1. $\Delta'; \Gamma' \vdash s : A$
2. $\Delta' \vdash \sigma\, x <: (\Delta\, x)[\sigma]$ *for all* x
3. $\Delta'; \Gamma' \vdash \tau\, x : (\Gamma\, x)[\sigma]$ *for all* x

Then $\Gamma; \Delta \vdash s[\sigma; \tau] : A[\sigma]$.

Proof. By induction on $\Delta'; \Gamma' \vdash s : A$

We require similar equational reasoning as in the context renaming lemma, using the context renaming lemma in the case of binders. □

Theorem 6.5 can then be shown by induction on $\Delta; \Gamma \vdash s : A$, using the previous context renaming and context substitution for typing of type applications and type abstraction respectively.

Discussion. See Figure 11 for an overview of the lines of code for the different challenges. Schäfer et al. [2015b] offer a detailed discussion. The recent proof shortened the code for Autosubst 1 slightly. The main gap to Needle&Knot lies in the proof of transitivity.

The proofs could be shortened for three reasons: First, compared to the proofs suggested in the appendix of the POPLMark challenge [Aydemir et al. 2005], parallel substitutions relieve us from many of the intermediate lemmas for single-point substitutions.

Second, a well-scoped syntax allowed us to remove the separate well-scopedness judgment. It also enables us to work with contexts in a more concise, functional fashion.

Third, vector substitutions omit interchanging lemmas for type and term renamings, respectively substitutions. Previous efforts could thus be simplified.

In short, vector substitutions and well-scoped terms allowed us to develop an even shorter and more elegant proof of part A of the POPLMark challenge.

There is still one area of improvement we want to tackle in the future. Note how – except for the main result – the main lemmas were all context renaming and context morphism lemmas. These lemmas can be obtained for free in a system based on HOAS. Extending Autosubst 2 in this way is currently work-in-progress [Schäfer and Stark 2018].

6.3 Call-by-push-value

A detailed description of our development can be found in [Forster et al. 2018]. We use this section to highlight some of the cases where Autosubst proved useful.

In our project, we consider three syntactic systems: A full λ-calculus (CBN), a fine-grained call-by-value variant which distinguishes between two mutually inductive sorts of terms and values (CBV), and call-by-push-value(CBPV, [Levy 1999]), which distinguishes between mutually inductive sorts of values and computations.

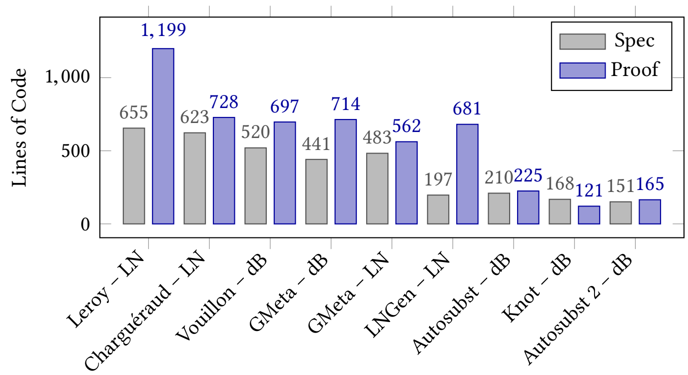

Figure 11. Comparison of Coq solutions to part A of the POPLMark challenge, measured using the coqwc utility.

Among others, CBPV contains an eliminator for pairs,

```
caseP : vl → (vl → vl → comp) → comp
```

which binds two variables at the same time. Neither this nor subsorts without variables, caused any problems.

Among other results, we provided translations from CBN and CBV to CBPV and proved simulation with respect to a weak and strong operational semantics. We proved the following substitution statement for CBV/CBN:

$$\overline{s}[\overline{\sigma}] = \overline{s[\sigma]}$$

where $\overline{s}$ and $\overline{\sigma}$ are the translation of CBV/CBN terms and substitutions to CBPV terms and substitutions, respectively. Again this required proving the corresponding statement for renamings first. In general, the CBPV project motivated us to switch to first-class renamings. In a first version, many statements needed manual changes to cope with renamings.

We used well-scoped syntax. This was especially pleasant as many results were technically involved, and well-scopedness ensured that we used the correct liftings. They further provide a way to express that certain results only hold for closed terms.

Finally, let us mention that the development stays faithful to the corresponding paper proofs and with Autosubst 2 binders caused no grief.

7 Related Work

Binders come with an enormous amount of literature.

POPLMark Challenge (Part A). It is fair to say, that the POPLMark challenge [Aydemir et al. 2005] re-sparked the interest in the correct handling of binders. There is a wealth of solutions, also in the Coq proof assistants [Aydemir and Weirich 2010; Keuchel et al. 2017; Lee et al. 2012; Pottier 2013;

Vouillon 2012]. A more detailed account can be found on the POPLMark website.

Autosubst and the σ-calculus. Autosubst 2 extends Autosubst [Schäfer et al. 2015a,b], a tool to generate support for parallel de Bruijn substitutions and the σ-calculus. The essential idea remains unchanged: on syntax with exactly one sort and one sort of variables both versions behave the same, except for the handling of renamings (Section 4). However, Autosubst 2's implementation is more flexible and allows a richer input syntax. Case studies as call-by-push-value or call-by-name variants would simply have not been supported by Autosubst 1.

Higher-order abstract syntax (HOAS). In HOAS [Pfenning and Elliott 1988], binders of the object language are represented as binders of the meta-language. Then, capture-avoiding substitution corresponds to function application and all constructs are automatically compatible with renamings and substitutions. However, HOAS cannot be implemented directly in Coq, since Coq's function spaces are extensional [Hofmann 1999b]. Weak HOAS [Despeyroux et al. 1995] or parametric HOAS [Chlipala 2008] avoid this problem with a slightly different definition, and still get compatibility with renamings.

Twelf [Pfenning and Schürmann 1999] or Beluga [Pientka and Dunfield 2010] directly implement variants of HOAS inside the proof assistant. They thus choose the trade-off of another function space and theory.

Code Generators. With Autosubst 2, we continue in the tradition of code generators. Ott and Lem [Mulligan et al. 2014; Sewell et al. 2007] provide tools to generate syntax for LaTeX and a wealth of proof assistants. Ott was later extended to locally nameless syntax [Aydemir and Weirich 2010], but we know of no interface for parallel de Bruijn substitutions and the σ-calculus.

Needle and Knot [Keuchel et al. 2017, 2016] generates code for unscoped, single-point de Bruijn substitutions. Autosubst 2 uses a similar (but simpler) intermediate layer and printing interface, while Needle and Knot do not compile their syntax to different proof assistants.

Our approach differs from an algebraic approach as in [Allais et al. 2018; Gheri and Popescu 2017]. There, instead of one (syntactic) sort for each object sort, there is *one* meta-sort of all syntactic systems parameterised by a signature. Although not so elegant, code generation comes with many practical advantages for the user. As each sort corresponds to an inductive type, we can use the immediate support of the proof assistant for inductive types. Moreover, we do not have to argue about instantiation of renamings or substitution which have no effect on the respective sort. Code generation also simplifies the handling of notation and automation.

Traversals. Syntax traversals [Allais et al. 2018, 2017] appear everywhere: Instantiation of renamings and substitutions are traversals, and even the substitution lemmas follow a similar structure. Moreover, traversals appear in various case studies as they allow a structured approach for recursion on higher-order types. A direct implementation of traversals in Autosubst 2 would simplify some of these proofs.

8 Conclusion and Future Work

We have outlined the theory and design of Autosubst 2, a 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 Bruijn term sorts, corresponding instantiation operations with vector substitutions, as well as a rewriting system for both unscoped and well-scoped syntax.

There are several directions for future work. First, we are interested in Autosubst 2 backends for Lean and Agda. This should be straightforward and is already work-in-progress.

From a theoretical point of view, we want to show that confluence and termination are preserved for both a σ-calculus with renamings and the multivariate σ-calculus. Moreover, we want to examine the matching problem for the σ-calculus. This would simplify the application of lemmas containing substitutions. Unfortunately, Coq does not allow to extend the matching algorithm directly.

We would further like to implement additional versions of automation, e.g., a variant without functional extensionality and a faster version of Autosubst's automation via reflection.

We are also interested in extending the input language of Autosubst 2. First to simply-typed structures and containers, then to *lists* or even more elaborated forms of binders (see e.g. [Keuchel and Jeuring 2012; Urban and Kaliszyk 2011]), needed, e.g., for part B of the POPLMark challenge, and last to possibly dependent predicates. More elaborated forms of binders might require us to change our input language from HOAS to a nominal variant.

Last, we would like to extend Autosubst 2 to syntax traversals, which provide renaming and substitution lemmas for structural recursive functions for free.

Acknowledgments

We thank the reviewers for their insightful comments and suggestions.

References

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

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).

Andreas Abel, Alberto Momigliano, and Brigitte Pientka. 2017. POPLMark Reloaded. In *Proceedings of the Logical Frameworks and Meta-Languages: Theory and Practice Workshop*.

Robin Adams. 2004. Formalized Metatheory with Terms Represented by an Indexed Family of Types. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs (TYPES'04)*. 1–16. https://doi.org/10.1007/11617990_1

Benedikt Ahrens and Julianna Zsido. 2011. Initial Semantics for Higher-Order Typed Syntax in Coq. *Journal of Formalized Reasoning* 4, 1 (2011).

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 90.

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.

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads need not be endofunctors. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 297–311.

Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *TPHOLs*, Vol. 3603. Springer, 50–65.

Brian E. Aydemir and Stephanie Weirich. 2010. *LNgen: Tool support for locally nameless representations.* Technical Report. University of Pennsylvania.

Andrew Cave and Brigitte Pientka. 2015. A case study on logical relations using contextual types. *arXiv preprint arXiv:1507.08053* (2015).

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, Vol. 43. ACM, 143–156.

Karl Crary. 2005. Logical relations and a case study in equivalence checking. *Advanced Topics in Types and Programming Languages* (2005), 223–244.

Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. 1996. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM (JACM)* 43, 2 (1996), 362–397.

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.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. 1995. Higher-order abstract syntax in Coq. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 124–138.

Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. 2018. Call-By-Push-Value in Coq: Operational, Equational, and Denotational Theory. *Submitted.* (2018). https://www.ps.uni-saarland.de/Publications/documents/ForsterEtAl_2018_Call-By-Push-Value.pdf

Lorenzo Gheri and Andrei Popescu. 2017. A formalized general theory of syntax with bindings. In *International Conference on Interactive Theorem Proving*. Springer, 241–261.

Healfdene Goguen and James McKinna. 1997. Candidates for substitution. *LFCS report series-Laboratory for Foundations of Computer Science ECS LFCS* (1997).

Martin Hofmann. 1995. Extensional concepts in intensional type theory. (1995).

Martin Hofmann. 1999a. Semantical analysis of higher-order abstract syntax. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on.* IEEE, 204–213.

Martin Hofmann. 1999b. Semantical analysis of higher-order abstract syntax. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on.* IEEE, 204–213.

Jonas Kaiser, Steven Schäfer, and Kathrin Stark. 2017a. Autosubst 2: Towards Reasoning with Multi-Sorted De Bruijn Terms and Vector Substitutions. In *Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '17)*. ACM, New York, NY, USA, 10–14. https://doi.org/10.1145/3130261.3130263

Jonas Kaiser, Steven Schäfer, and Kathrin Stark. 2018. Binder aware recursion over well-scoped de Bruijn syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 293–306.

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

Steven Keuchel and Johan T Jeuring. 2012. Generic conversions of abstract syntax representations. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming*. ACM, 57–68.

Steven Keuchel, Tom Schrijvers, and Stephanie Weirich. 2017. *Needle & Knot: Boilerplate bound tighter.* Technical Report.

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder boilerplate tied up. In *European Symposium on Programming Languages and Systems*. Springer, 419–445.

Gyesik Lee, Bruno C.D.S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. 2012. GMeta: A Generic Formal Metatheory Framework for First-Order Representations. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 7211. Springer Berlin Heidelberg, 436–455.

Paul Blain Levy. 1999. Call-by-push-value: A subsuming paradigm. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 228–243.

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-world Semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 175–188. https://doi.org/10.1145/2628136.2628143

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 23.

Ulf Norell. 2008. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*. Springer, 230–266.

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.

Frank Pfenning and Carsten Schürmann. 1999. System description: Twelf – A meta-logical framework for deductive systems. In *International Conference on Automated Deduction*. Springer, 202–206.

Brigitte Pientka and Joshua Dunfield. 2010. Beluga: A framework for programming and reasoning with deductive systems (system description). In *International Joint Conference on Automated Reasoning*. Springer, 15–21.

Andrew M Pitts. 2013. *Nominal sets: Names and symmetry in computer science.* Cambridge University Press.

François Pottier. 2013. DBLIB, a Coq library for dealing with binding using de Bruijn indices. https://github.com/fpottier/dblib.

Steven Schäfer, Gert Smolka, and Tobias Tebbi. 2015a. Completeness and Decidability of de Bruijn Substitution Algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*. Springer-Verlag, Berlin, Heidelberg, 67–73. https://doi.org/10.1145/2676724.2693163

Steven Schäfer and Kathrin Stark. 2018. Embedding Higher-Order Abstract Syntax in Type Theory. In *Abstract for Types Workshop*.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015b. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 359–374. https://doi.org/10.1007/978-3-319-22102-1_24

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: Effective tool support for the working semanticist. *ACM SIGPLAN Notices* 42, 9 (2007), 1–12.

Christian Urban and Cezary Kaliszyk. 2011. General bindings and alpha-equivalence in Nominal Isabelle. In *European Symposium on Programming*. Springer, 480–500.

Jérôme Vouillon. 2012. A Solution to the POPLmark Challenge Based on de Bruijn Indices. *Journal of Automated Reasoning* 49, 3 (2012), 327–362.

Philip Wadler. 2003. A prettier printer. *The Fun of Programming, Cornerstones of Computing* (2003), 223–243.