Sulfur: a Reflective Tactic for Substitution Simplification

Mathis Bouverot-Dupuis, Théo Winterhalter, Kathrin Stark, Kenji Maillard

ACM Reference Format:
Mathis Bouverot-Dupuis, Théo Winterhalter, Kathrin Stark, Kenji Maillard . 2026. Sulfur: a Reflective Tactic for Substitution Simplification. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

When formalizing the meta-theory of programming languages or type systems in proof assistants such as Rocq [23], a key design decision is the choice of variable representation (see e.g. the PoplMark challenge [4]). De Bruijn indices [7] are a popular option because they make α-equivalence coincide with syntactic equality. However, de Bruijn indices also introduce significant overhead in the form of lift and renaming operations, which require many technical lemmas and can make proofs tedious.

To address this, many libraries attempt to automate repetitive aspects of dealing with de Bruijn indices and substitution, both in programming languages (e.g. Rebound [8] or BindLib [14]) and in proof assistants (e.g. Autosubst [19, 22], Tealeaves [9], DBGen [16], Fiore and Szamozvancev [10], and Allais et al. [3]). In particular, while Autosubst has been successfully used in many formalizations [2, 6, 11, 12, 21, 24, 25], its simplification tactic `asimpl` suffers from significant performance issues when used in large developments. In this talk, we present Sulfur[1] (Substitution logical framework using reflection), a Rocq plugin that attempts to solve the performance issues of Autosubst's `asimpl` tactic by implementing it as a *reflective tactic* [5, 13, 15]. For now, Sulfur supports single-sorted, extrinsic syntax: extensions to more complex signatures are discussed in Section 4.

[1] Our development is available at https://github.com/MathisBD/rocq-sulfur.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Using Sulfur

Sulfur provides a similar interface to Autosubst: given a user-specified language signature (Figure 1a), Sulfur automatically generates Rocq code (Figure 1b) implementing an inductive type `term` representing terms with variables encoded as de Bruijn indices, and a parallel substitution function.

```
Sulfur Generate {{
  term : Type
  app : term → term → term
  lam : (bind term in term) → term
}}.
```

(a) User-specified signature.

```
Inductive term :=
| var (i : nat)
| app (t u : term)
| lam (t : term).

Definition substitute : (nat → term) → term → term.
(** substitute s t is abbreviated as t[s]. *)
```

(b) Code generated by Sulfur.

```
Lemma technical_lemma (t1 t2 : term) (s : nat → term) :
  t1[0 . (s >> shift)][t2[s] . id] = t1[t2 . id][s].
```

(c) Example substitution-heavy lemma.

Figure 1. Untyped λ-calculus, using Sulfur. The code in Figure 1b is automatically generated by Sulfur.

Most importantly, Sulfur automates reasoning about substitution: like Autosubst, it provides a tactic `asimpl` which simplifies terms and substitutions according to the rules of σ-calculus [1]. For instance consider the technical lemma in Figure 1c, which is needed when proving standard properties of λ-calculus. In this lemma, `id` is the identity substitution, `shift` is the substitution which adds 1 to every de Bruijn index, `t . s` is a substitution which maps index `0` to `t` and index `i+1` to `s i`, and `s1 >> s2` is the composition of `s1` followed by `s2`. Proving this equality by hand requires significantly more effort than one might expect, using many auxiliary lemmas. However, `asimpl` simplifies both sides of the equation to `t1[t2[s] . s]`, trivializing the proof.

3 Key ideas

While Sulfur provides, by design, the same user interface as Autosubst, its implementation diverges significantly. In particular, Autosubst's `asimpl` tactic relies on Rocq's *setoid rewrite* facilities [20], and has significant performance issues when used in large developments. Sulfur aims to improve performance by using a logical framework approach: we define a generic notion of syntax with explicit substitutions within Rocq, and implement `asimpl` as a plain Rocq function over this generic syntax.

Signatures. We encode the *signature* of a language as a set of constructors (`ctor`) along with information about the arity and binding structure of each constructor (`ctor_args`):

```
Inductive arg :=
| arg_term
| arg_bind (x : arg).
```

```
Record signature := {
  ctor : Type ;
  ctor_args : ctor -> list arg }.
```

As an example we give the signature for the untyped lambda-calculus of Figure 1:

```
Inductive ctor := App | Lam.
ctor_args App = [ arg_term ; arg_term ]
ctor_args Lam = [ (arg_bind arg_term) ]
```

Generic syntax. Inspired by σ-calculus, we define a notion of syntax with explicit substitutions, which is moreover *generic*, i.e. parameterized over a signature. We give a simplified version of generic syntax:

```
Inductive g_term {s : signature} :=
| g_var (i : nat)
| g_ctor (c : ctor s) (args : list g_term)
| g_substitute (s : g_subst) (t : g_term)
| g_term_mvar (m : mvar)
with g_subst {s : signature} :=
| g_id
| g_shift
| g_cons (t : g_term) (s : g_subst)
| g_comp (s1 s2 : g_subst)
| g_subst_mvar (m : mvar).
```

Substitutions are not arbitrary functions of type `nat → g_term` but are instead built using a set of constructors `gid`, `gshift`, `gcons`, and `gcomp`, which correspond to the substitution primitives `id`, `shift`, `_ . _`, and `_ >> _` of the σ-calculus. Substitutions can be explicitly applied to terms using `g_substitute`.

Not all substitutions `nat → term` are representable using the constructors of σ-calculus. Substitutions which don't fit in the framework of σ-calculus are represented using *meta-variables* (constructor `g_subst_mvar`). Terms can similarly contain meta-variables (constructor `g_term_mvar`). Meta-variables `mvar` are drawn from an infinite set with decidable equality (in our development we define `mvar` as `nat`).

Generic syntax contains enough information to be able to implement a simplification function directly in Rocq:

```
Definition simplify sig : g_term sig → g_term sig.
```

Reification and denotation. Generic syntax is quite far from what we picture as the untyped λ-calculus, and we certainly do not want users to work with `g_term`. Thus, we still generate syntax specialized to the user's signature, as in Figure 1b.

The mapping between user syntax and generic syntax, i.e. between `term` and `g_term`, is fairly straightforward. A *denotation* function `denote : env → gterm sig → term` can be implemented directly within Rocq (for any signature `sig`) by simple structural recursion over the input term. The first argument of `denote` is an *environment*, which is a mapping from meta-variables to concrete terms and substitutions.

The other direction, *reification*, requires us to step outside Rocq. Leveraging Rocq's support for meta-programming, we can reify a term `t` into a generic term `t' : g_term sig` and an environment `e : env`, which are required to obey the invariant that `denote e t'` is convertible (i.e. definitionally equal) to `t`.

Implementing `asimpl` using reflection. In order to simplify terms in user syntax, we first establish the soundness of `simplify`:

```
Theorem soundness e t : denote e t = denote e (simplify t).
```

Using all these ingredients, we can implement `asimpl` as follows. To simplify a term `t : term` appearing in a goal:

1. Reify `t` into `t' : g_term sig` and `e : env`.
2. Because `t` is convertible to `denote e t'`, which itself is equal to `denote e (simplify t')`, we can replace all occurrences of `t` with `denote e (simplify t')` in the goal.
3. Use Rocq's evaluation mechanisms to reduce the expression `denote e (simplify t')`.

4 Future work

Benchmarking. Early experiments suggest that Sulfur's `asimpl` tactic is indeed faster than the equivalent Autosubst tactic. We plan on conducting detailed benchmarks to quantify the performance gain more precisely.

More complex signatures. Central future work is to extend Sulfur to accommodate more complex signatures, scaling up to the full generality of Autosubst. For instance, signatures with multiple sorts of terms (e.g. System F) and signatures including functors (e.g. using lists to represent n-ary applications) are not supported yet. Extending our generic syntax to handle multiple sorts requires encoding subtle invariants: for instance in System F, term variables cannot occur in types and thus parallel substitution in types only requires a substitution on type variables. We hope to benefit from related work on multi-sorted substitution [17].

Proving completeness. The following completeness theorem holds in simpler variants of σ-calculus [18]:

```
Theorem completeness (t t' : g_term sig) :
  (forall e, denote e t = denote e t') →
  simplify t = simplify t'
```

Intuitively this states that reification followed by simplification is enough to decide equality of concrete terms. Unfortunately, this completeness theorem does not hold on our version of generic syntax due to the presence of explicit renamings, however we conjecture that some weaker version of completeness still holds, and believe that proving such a result is an interesting direction for future work.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

[2] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. https://doi.org/10.1145/3636501.3636951

[3] 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. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (July 2018), 30 pages. https://doi.org/10.1145/3236785

[4] 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 *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics* (Oxford, UK) *(TPHOLs'05)*. Springer-Verlag, Berlin, Heidelberg, 50–65. https://doi.org/10.1007/11541868_4

[5] Samuel Boutin. 1997. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, Martín Abadi and Takayasu Ito (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 515–529.

[6] Liron Cohen, Ariel Grunfeld, Dominik Kirst, and Étienne Miquey. 2025. Syntactic Effectful Realizability in Higher-Order Logic. *CoRR* abs/2506.09458 (2025). https://doi.org/10.48550/ARXIV.2506.09458 arXiv:2506.09458

[7] N.G 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. https://doi.org/10.1016/1385-7258(72)90034-0

[8] Noé De Santo and Stephanie Weirich. 2025. Rebound: Efficient, Expressive, and Well-Scoped Binding. In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium* (Singapore, Singapore) *(Haskell '25)*. Association for Computing Machinery, New York, NY, USA, 38–52. https://doi.org/10.1145/3759164.3759348

[9] Lawrence Dunn, Val Tannen, and Steve Zdancewic. 2023. Tealeaves: Structured Monads for Generic First-Order Abstract Syntax Infrastructure. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:20. https://doi.org/10.4230/LIPIcs.ITP.2023.14

[10] Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.* 6, POPL, Article 53 (Jan. 2022), 29 pages. https://doi.org/10.1145/3498715

[11] Yannick Forster, Dominik Kirst, and Dominik Wehr. 2021. Completeness theorems for first-order logic analysed in constructive type theory: Extended version. *Journal of Logic and Computation* 31, 1 (01 2021), 112–151. https://doi.org/10.1093/logcom/exaa073 arXiv:https://academic.oup.com/logcom/article-pdf/31/1/112/36719784/exaa073.pdf

[12] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *Proc. ACM Program. Lang.* 4, ICFP, Article 114 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408996

[13] Benjamin Grégoire and Assia Mahboubi. 2005. Proving equalities in a commutative ring done right in coq. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics* (Oxford, UK) *(TPHOLs'05)*. Springer-Verlag, Berlin, Heidelberg, 98–113. https://doi.org/10.1007/11541868_7

[14] Rodolphe Lepigre and Christophe Raffalli. 2018. Abstract representation of binders in ocaml using the bindlib library. *arXiv preprint arXiv:1807.01872* (2018).

[15] Gregory Malecha, Adam Chlipala, and Thomas Braibant. 2014. Compositional Computational Reflection. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 374–389.

[16] Emmanuel Polonowski. 2013. Automatically Generated Infrastructure for De Bruijn Syntaxes. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 402–417.

[17] Hannes Saffrich. 2024. Abstractions for Multi-Sorted Substitutions. In *15th International Conference on Interactive Theorem Proving (ITP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 309)*, Yves Bertot, Temur Kutsia, and Michael Norrish (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2024.32

[18] 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* (Mumbai, India) *(CPP '15)*. Association for Computing Machinery, New York, NY, USA, 67–73. https://doi.org/10.1145/2676724.2693163

[19] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015 (LNAI)*, Xingyuan Zhang and Christian Urban (Eds.). Springer-Verlag.

[20] Matthieu Sozeau. 2009. A new look at generalized rewriting in type theory. *Journal of formalized reasoning* 2, 1 (2009), 41–62.

[21] Simon Spies and Yannick Forster. 2020. Undecidability of higher-order unification formalised in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 143–157. https://doi.org/10.1145/3372885.3373832

[22] 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* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 166–180. https://doi.org/10.1145/3293880.3294101

[23] The Coq Development Team. 2024. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.14542673

[24] Dawit Tirore, Jesper Bengtson, and Marco Carbone. 2023. A Sound and Complete Projection for Global Types. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:19. https://doi.org/10.4230/LIPIcs.ITP.2023.28

[25] Théo Winterhalter. 2024. Dependent Ghosts Have a Reflection for Free. *Proc. ACM Program. Lang.* 8, ICFP, Article 258 (Aug. 2024), 29 pages. https://doi.org/10.1145/3674647