Sulfur: Automating Substitution-preserving Syntaxes and Judgments in Rocq

Théa Hervier
ENS de Lyon

Mathis Bouverot-Dupuis
INRIA Paris, ENS Paris

Thiago Felicissimo
INRIA Rennes

Théo Winterhalter
Université Paris-Saclay, INRIA, CNRS, ENS Paris-Saclay, LMF

I. Introduction

Formalising the meta-theory of a programming language in a proof assistant such as Rocq [1] requires choosing a representation for variables to avoid suffering down the line (see the POPLMark challenge [2]). The most popular solution is likely de Bruijn indices [3] which make α-equivalence coincide with syntactic equality. Great for machines, not so much for humans who tend to make off-by-one mistakes when manipulating indices by hand.

Users thus turn to tools to deal with de Bruijn indices automatically, generating boilerplate definitions such as substitution. Notable mentions are Autosubst [4, 5], TeaLeaves [6], DBGen [7], Fiore and Szamozvancev [8], Allais *et al.* [9]. Besides generating useful lemmas about substitution, Autosubst offers a tactic `asimpl` to simplify expressions mixing terms and substitutions, and recent work on the Sulfur project (Substitution logical framework using reflection) [10] focused on making this tactic more efficient by leveraging reflection.

While this is already a huge step forward when formalising languages with binders, another recurring task is proving that judgments such as typing relations are preserved by substitution. For instance, for a dependent type theory, one typically defines a typing judgment $\Gamma \vdash t : A$ and then shows that typing is preserved by substitution: if we have a substitution giving a term in Δ of the right type for every variable of Γ, written $\Delta \vdash \sigma : \Gamma$, then $\Delta \vdash t[\sigma] : A[\sigma]$. In our own experience, these proofs tend to always be awfully similar, and we often start by copying and pasting the proofs from earlier projects [11–14].

Thankfully, such proofs could in principle also be automated by taking inspiration from the literature. Indeed, works like Uemura's SOGATs [15], Haselwarter and Bauer's finitary type theories [16, 17] and Felicissimo's bidirectional framework [18] all provide general definitions of type theories, for which they prove basic properties like stability under weakening and substitution once and for all, by quantifying over all theories supported by their frameworks.

Contributions. Taking inspiration from these works, we extend Sulfur to generate the substitution lemma generically for a presentation of *judgments*. We do so by defining in Rocq[1] a type of signature for judgments, comprised of rules over a previously defined Sulfur syntax. Compared with the aforementioned work, our notion of judgment is more flexible and supports a wider range of cases, such as reduction or untyped conversion, thus offering more freedom to users.

II. Sulfur in Practice

Sulfur provides a similar interface to Autosubst. The user writes a high-level description of their syntax as follows:

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

It then generates (via a meta-program) an inductive type `tm` representing terms with variables encoded as de Bruijn indices, a substitution function, and many lemmas about it:

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

```
Definition substitute :
  (nat → tm) →
  tm → tm.
```

We now seek to extend this workflow to also generate substitution lemmas for various judgments (e.g. reduction, conversion, typing). While our main contribution is about the technical machinery needed to automate such proofs (which we detail in the next section) we give a brief overview of our goal interface, which we have yet to implement. As an example, the user should be able to additionally specify e.g. a typing judgment `typ` with some rules (here `typ_abs`):

```
Sulfur Generate {{
  typ : list type → tm → type → Prop
  typ_abs : typ [A] b B → typ [] (lam b) (A → B)
  (* More rules... *)
}}.
```

Sulfur would then generate an inductive type encoding the typing judgment, along with its substitution lemma. We are currently investigating the best syntax for user input: the one above has the drawback that it exposes technicalities of our representation such as the implementation of contexts and the fact that we require it to be empty in the conclusion of a rule. The talk will be an opportunity to gather feedback on this.

III. Rocq implementation

Following the Sulfur approach to syntax, we develop a logical framework parameterised over (an encoding of) judgments and rules. Under reasonable assumptions on the rules, we prove a generic substitution lemma for the judgments.

[1]https://github.com/TheaHervier/rocq-sulfur/tree/judgments

A. Encoding judgments

Our goal is to encode relations of the form `Γ ⊢ j` where `Γ` is a context and `j` is a judgment. The form of judgments and contexts is chosen by the user. For instance, the typing judgments of the λΠ-calculus are pairs of terms, while contexts are lists of terms. In general, the arity of a judgment is represented by a user-defined functor `jdgF : Type → Type` that will layer be instantiated by the syntax. In the case of typing for λΠ, we thus use `jdgF X := X * X` because on the right of the turnstile we have two terms: the term and its type.

Contexts are represented as lists of *declarations*, where the declaration at position `n` represents an information about variable `n`. Similarly to judgments, declarations are represented with a functor `declF : Type → Type`, and contexts are represented by the composition of the functor `list` and `declF`. In λΠ, we choose `declF X := X`, we hold a type for each variable and nothing more.

Derivability is then defined as a relation of type:

```
list (declF term) → jdgF term → Prop
```

B. Signature of a judgment

Users characterise a judgment by providing their signature. It is given by the functors `jdgF` and `declF` above, together with a list of rules. Take for instance the conversion rule of λΠ:

$$\frac{\Gamma \vdash t : A \qquad A \equiv A' \qquad \Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash t : A'}$$

We consider here a presentation with untyped conversion, hence the absence of context on the conversion premise.

When representing this rule, we will actually strip the Γ prefix of all contexts, following Bauer *et al.* [19], meaning the conclusion has to be in the empty context. A rule is thus given by a conclusion, here $t : A'$, as well as a list of premises. The first premise $\cdot \vdash t : A$ is an *inductive premise*, referring to the judgment we are currently defining. It consists in a context extension (here, the empty context) and a judgment $t : A$. The second premise $A \equiv A'$ is a *predicate premise*, which can reuse anything that was already defined before (e.g. conversion), including (but not limited to) relations generated by Sulfur, provided they respect substitution.

In order to apply a rule, one must instantiate it: in the rule above, one must explain what are the terms t, A and A'. This is why, while relations talk about `term`, signatures use another type of terms with *meta-variables*.

C. Terms with meta-variables

We extend Sulfur's `term` type to `mterm` by adding meta-variables applied to a (finite) substitution:

```
Inductive term := E_var (i : nat)
| E_ctor (c : ctor) (a : list term).

Inductive mterm :=
| M_var (i : nat)
| M_ctor (c : ctor) (a : list mterm)
| M_mvar (k : nat) (σ : list mterm)
```

We then define an evaluation function that instantiates all meta-variables of an `mterm` in order to produce a `term`:

```
Fixpoint inst (m : nat → term) : mterm → term.
```

Note that there a number of commutation lemmas between substitution and instantiation that we show once and for all.

D. Variable case

Up till now we did not mention rules involving variables. They are treated as a special case. The idea is that judgments associated to a variable should *derive* from the context. In the case of λΠ, the declaration A is associated to the judgment form $- : A$, where the hole is filled by a term (a variable or something to substitute it by). We represent this with a map:

```
make_jdg : declF term → term → jdgF term
```

If the declaration `d` is at position `n` in `Γ`, we deduce `Γ ⊢ make_jdg d (E_var n)`. This allows us to retrieve the information on variables from the declarations in the context.

E. Preservation by substitution

Given a substitution `σ : nat → term`, we define a standard subtitution typing judgment $\Delta \vdash_s \sigma : \Gamma$ inductively by:

$$\frac{}{\Delta \vdash_s \sigma : \cdot} \qquad \frac{\Delta \vdash_s \uparrow \sigma : \Gamma \qquad \Delta \vdash \texttt{make\_jdg}\ (A[\uparrow \sigma])(\sigma\ 0)}{\Delta \vdash_s \sigma : \Gamma, A}$$

The declaration $A[\uparrow \sigma]$ is the declaration `A : declF term` in which we substituted the lifted substitution $\uparrow \sigma$ functorially. The premise $\Delta \vdash$ `make_jdg` $(A[\uparrow \sigma])\ (\sigma\ 0)$ means that in context Δ, the term $\sigma\ 0$ satisfies the judgment induced by declaration $A[\uparrow \sigma]$.

We finally prove that the relation we define with a signature preserves substitution:

```
Theorem preserve_subst Γ j Δ σ :
  Γ ⊢ j → Δ ⊢s σ : Γ → Δ ⊢ j[σ].
```

where `j[σ]` is the judgment `j` where we functorially applied substitution `σ`.

The proof of this theorem follows the exact same structure as in the examples mentioned in Section I. Of course the theorem only holds for signatures which obey a few (mostly administrative) assumptions, the most important one being that the predicate premises of every rule are compatible with substitution. They were not an obstracle to building examples.

F. Mutually defined judgments

Our notion of judgement is rather compositional, as we mentioned before it supports referring to previously defined judgments, without requiring that they were derived by Sulfur. We in fact additionally support mutually defined judgments (although we kept things simple in the presentation before). We achieve this by indexing the signature by a finite type. Thanks to this extension, we are able to support both versions of λΠ, either by defining first untyped conversion and then typing on top of it, or by defining conversion and typing mutually, enabling the support of η laws.

IV. Future work

The next step is to implement the meta-programming glue so that the syntax we presented in Section II becomes a reality. Just like Sulfur already does for syntax and substitution, we would like to hide implementation details and provide a nice interface with an inductive type without having to go through the encoding we use internally. Once we achieve this, it will be much easier to explore various judgments and potentially extend Sulfur to support more if our experiments suggest they are needed. We were thinking for instance of polarised calculi.

References

[1] The Rocq Development Team, The Rocq Prover, [Online]. Available: https://doi.org/10.5281/zenodo.17473943, 2025.

[2] Aydemir *et al.*, "Mechanized metatheory for the masses: the PoplMark challenge," in, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, , pp. 50–65, 2005, Oxford, UK: Springer-Verlag.

[3] de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," *Indagationes Mathematicae (Proceedings)*, , vol. 75, no. 5, pp. 381–392, 1972.

[4] Schäfer *et al.*, "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions," in, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, , Aug. 2015, Springer-Verlag.

[5] Stark *et al.*, "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*, , pp. 166–180, 2019, Cascais, Portugal: Association for Computing Machinery.

[6] Dunn *et al.*, "Tealeaves: Structured Monads for Generic First-Order Abstract Syntax Infrastructure," in, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, , pp. 14:1–14:20, 2023, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[7] Polonowski, "Automatically Generated Infrastructure for De Bruijn Syntaxes," in, *Interactive Theorem Proving*, , pp. 402–417, 2013, Berlin, Heidelberg: Springer Berlin Heidelberg.

[8] Fiore and Szamozvancev, "Formal metatheory of second-order abstract syntax," *Proc. ACM Program. Lang.*, , vol. 6, no. POPL, Jan. 2022.

[9] Allais *et al.*, "A type and scope safe universe of syntaxes with binding: their semantics and proofs," *Proc. ACM Program. Lang.*, , vol. 2, no. ICFP, July 2018.

[10] Bouverot-Dupuis *et al.*, "Sulfur: a Reflective Tactic for Substitution Simplification," in, *RocqPL 2026: Rocq for Programming Languages*, , 2026.

[11] Winterhalter, "Dependent Ghosts Have a Reflection for Free," *Proceedings of the ACM on Programming Languages*, , no. 258, pp. 630–658, Aug. 2024.

[12] Leray and Winterhalter, "Encode the Cake and Eat it Too," *Proceedings of the ACM on Programming Languages*, , vol. 10, no. 62, Jan. 2026.

[13] Felicissimo *et al.*, "Definitional Proof Irrelevance Made Accessible," in, *LICS 2026 - 41st Annual Symposium on Logic in Computer Science*, , July 2026, Lisbon, Portugal.

[14] Felicissimo and Winterhalter, Confluence Techniques for Dependent Type Theory with Typed Conversion, [Online]. Available: https://inria.hal.science/hal-05520710, 2026.

[15] Uemura, *Abstract and concrete type theories*, Doctoral dissertation, 2021.

[16] Bauer and Komel, "An Extensible Equality Checking Algorithm for Dependent Type Theories," *Logical Methods in Computer Science*, , vol. 18, no. 1, 2022.

[17] Haselwarter and Bauer, "Finitary Type Theories With and Without Contexts," *Journal of Automated Reasoning*, , vol. 67, no. 4, p. 36, 2023.

[18] Felicissimo, *Generic Bidirectional Typing in a Logical Framework for Dependent Type Theories*, Doctoral dissertation, 2024.

[19] Bauer *et al.*, "A General Definition of Dependent Type Theories," *arXiv preprint arXiv:2009.05539*, , 2020.