Work-in-Progress: A Tactic for Pattern Matching in Autosubst

Mathews George
Heriot-Watt University
Edinburgh, UK
mg2079@hw.ac.uk

Kathrin Stark
Heriot-Watt University
Edinburgh, UK
k.stark@hw.ac.uk

Autosubst enables automatic equality-checking up to the σ-calculus for assumption-free equalities, allowing users to avoid cumbersome reasoning about de Bruijn indices. While effective in many cases, this approach is inapplicable when matching against typing rules, reduction relations, or lemmas, requiring users to either phrase typing rules in a way that they work with Autosubst or even stating explicitly an alternative de Bruijn term. But even without β-reduction, solutions of matching may not be unique. This paper presents a work-in-progress method for automatically pattern matching against assumptions, evaluated on standard case studies including the POPLMark and POPLMark Reloaded challenges.

1 Introduction

De Bruijn terms are a canonical technique for representing terms with binders [11], and have a long history of being used in mechanised formalisations of languages with binders in proof assistants. Despite this, they are often as regarded as cumbersome in practice [6], largely due to the overhead involved in proving and applying the many technical lemmas required for substitution and renaming.

The Autosubst code generator [33, 36] addresses this issue by automating reasoning about de Bruijn terms in the Rocq (formerly Coq) proof assistant in many cases. Given a signature describing a custom syntax, Autosubst automatically proves the substitution lemmas corresponding to the equations in the σ-calculus [1]. Central to the approach, Autosubst comes with a simplification tactic called `asimpl` which normalizes terms according to these equations. As the de Bruijn algebra forms a sound and complete model of the σ-calculus [32], and as the rewriting system is convergent [10], equality in the de Bruijn algebra is decidable by this tactic. In summary, this allows users to work with de Bruijn syntax without manually applying many technical lemmas.

However, this approach becomes less effective in situations where we want to unify a goal with an assumption, in Rocq's case via the `apply` or `eapply` tactic. In particular, Autosubst's automation is designed for assumption-free reasoning and does not support matching modulo σ-equivalence. As a consequence, the user must compensate for these limitations in their proof scripts. Common workarounds include stating rules in an indirect form to facilitate matching, introducing auxiliary lemmas with the right instantiations, and manually transforming goals during proofs to make them applicable, together with directly defining the values of appearing existential variables.

A representative example is the formulation of the type application rule in System $F_{<:}$ in the POPLMark solution based on Autosubst:

Submitted to:
SOS 2007

$$T\_Tapp : \forall\ \Delta\ \Gamma\ s\ A\ A'\ B\ B',\ \Delta;\Gamma \vdash s : \forall A.B \to\ \Delta \vdash A' <: A \to\ B' = B[A' \cdot id] \\ \to \Delta;\Gamma \vdash s\ A' : B'$$

Here, the result type is expressed using a fresh meta-variable B' rather than the more direct expression $B[A' \cdot id]$. While this formulation enables the use of the `eapply` tactic, it shifts the burden to the user, who (when applying this rule to a goal) must subsequently discharge the additional equality premise. This premise typically contains existential variables, and the `asimpl` tactic is only helpful in the case that the user can give the value of these existential variables upfront.

For example, proving the context morphism lemma in the POPLMark challenge, in the corresponding case we get stuck with the following goal

$$\Delta;\Gamma \vdash s[\sigma;\tau]\ A'[\sigma] : B[A'[\sigma] \cdot \sigma]$$

where, when applying `T_Tapp`, we need to solve the equation $B[A[\sigma] \cdot \sigma] =?B[A[\sigma] \cdot id]$ with a constraint on $?B$ as $\Delta;\Gamma \vdash s[\sigma] : \forall A[\sigma].?B$.

Such indirect formulations are pervasive. They arise not only in typing rules, but also in reduction relations [14, 2], weakening and renaming lemmas, and even in mechanisations of dependent type theory [3]. Even when rules are stated directly, auxiliary lemmas are often introduced solely to make them compatible with `apply`.

This raises a natural question: can we recover the level of automation provided by Autosubst while supporting matching modulo σ-equivalence in the presence of assumptions?

The problem is inherently difficult. Moura et al. showed that Dowek et al.'s method [12] does not decide second-order matching for the full $\lambda\sigma$-calculus [25], but even without β, solutions are not necessarily unique. In practice, this means that a matching problem may admit multiple candidates, not all of which satisfy the premises in the context.

Example 1 (Matching up to σ-equivalence is not unique.)**.** The σ-matching problem we need to solve is constrained by the premises in which the unknown variables appear. In the above example, we need to solve the equation $B[A[\sigma] \cdot \sigma] =?B[A[\sigma] \cdot id]$ with a constraint on $?B$ as $\Delta;\Gamma \vdash s[\sigma] : \forall A[\sigma].?B$. We have at least two solutions for $?B$, $B[\Uparrow\ \sigma]$ and $B[A[\sigma \circ \uparrow] \cdot \sigma \circ \uparrow]$. The latter solution does not hold with the premise.

As a consequence, any practical solution must rely on heuristics. In the worst case, an adversarial example can always be constructed where a heuristic selects an incorrect solution, even when a correct one exists.

Contributions. We extend the Autosubst framework with a pattern-matching tactic modulo σ-equivalence. Our main contributions are as follows:

1. We introduce a heuristic tactic `as_apply`, intended as a replacement for the `apply/eapply` tactic in Autosubst-based developments.
2. We generalize this tactic to support the full input language of Autosubst.
3. We evaluate our approach on several case studies, including the complete POPLMark and POPLMark Reloaded Challenge and show that despite its theoretical limitations, first results suggest it performs well in practice.

The development including all results is available online [17].

$$\boxed{\begin{array}{r}(s\ t)[\sigma] \equiv s[\sigma]\ t[\sigma] \\ (\lambda.s)[\sigma] \equiv \lambda.(s[\Uparrow \sigma]) \\ s[\sigma][\tau] \equiv s[\sigma \circ \tau] \\ (s \cdot \sigma) \circ \tau \equiv\ s[\tau] \cdot \sigma \circ \tau \\ (\sigma \circ \tau) \circ \rho \equiv \sigma \circ \tau \circ \rho\end{array}} \qquad \begin{array}{r} s[id] \equiv s \\ \sigma \circ id \equiv \sigma \\ id \circ \sigma \equiv \sigma \\ 0 \cdot \uparrow \equiv id \end{array} \qquad \begin{array}{r} 0[s \cdot \sigma] \equiv s \\ \uparrow \circ\ (s \cdot \sigma) \equiv \sigma \\ 0[\sigma] \cdot \uparrow \circ\ \sigma \equiv \sigma \end{array}$$

Figure 1: The rewriting system of the σ-calculus

2 Background

We consider de Bruijn terms for the λ-calculus [11]:

$$s,t \in \mathbb{T}\ ::=\ n \mid s\ t \mid \lambda.s \qquad (n \in \mathbb{N})$$

where $s\ t$ denotes application, $\lambda.s$ abstraction, and n are de Bruijn indices which serve as variables bound by abstraction.

Parallel substitutions σ,τ are total functions mapping indices to terms. Intuitively, a substitution can be seen as an infinite sequence of terms. Sometimes, we need to deal with a special kind of substitutions, written ξ,ζ, that map indices to indices known as *renamings*.

A *de Bruijn algebra* is a first-order two-sorted algebra formed of de Bruijn terms and substitutions, and certain constants and operations from the σ-calculus of Abadi et al. [1]. The central operation is instantiation: $s[\sigma]$ replaces the free indices of the term s with the terms provided by the substitution σ.

The σ-calculus comes with a defined set of primitives. The cons operation $s \cdot \sigma$ prepends a term s to a substitution σ. The composition of substitutions is defined as $(\sigma \circ \tau)\ n\ := (\sigma\ n)[\tau]$. We further have constant substitutions such as the identity $id\ := \lambda n.n$ and the shift $\uparrow\ := \lambda n.(n+1)$. The lifting of a substitution is defined as and used as notation for $\Uparrow\ \sigma := 0 \cdot \sigma \circ \uparrow$.

The σ-calculus is equipped with a directed set of equational rules over these primitives (Figure 1). We denote $s \equiv_{\sigma'} t$ if s and t are provably equal by rewriting with a fragment σ' of σ-rules. We denote by $\sigma_{\texttt{min}}$ the fragment consisting of the rules inside the box in Figure 1, which characterise instantiation and its compositionality properties.

Schäfer et al. showed that the de Bruijn algebra forms a sound and complete model of the σ-calculus [32], and, as the rewriting system induced by these rules is known to be convergent [10], equality in the de Bruijn algebra is decidable by normalisation with respect to the σ-rules.

We also require notions from σ-matching [12, 7]. A σ-substitution θ is a function from the object variables to σ-expressions such that $\theta\ x \neq x$ for finitely many x. Its domain is $\mathcal{D}(\theta) := \{x \mid \theta\ x \neq x\}$. We write θ as $\{x_1 \mapsto s_1, ..., x_n \mapsto s_n\}$ where $x_i \in \mathcal{D}(\theta)$ and denote its application to a σ-expression s simply as $\theta\ s$. Two substitutions θ_1 and θ_2 are σ'-equal iff $\theta_1\ x \equiv_{\sigma'} \theta_2\ x$ for all x. Otherwise, θ_1 and θ_2 are σ'-different.

We distinguish between *bound* and *free* object variables. The free variables $\mathcal{F}(s)$ are the ones we need to resolve in a unification problem [12]. We use the notation $s =_{\sigma'} s'$ to denote a matching equation in a fragment σ' of the σ-calculus where s does not have free variables. We say $s =_{\sigma'} t$ has a solution θ iff $s \equiv_{\sigma'} \theta\ t$ and $\mathcal{D}(\theta) \subseteq \mathcal{F}(t)$. We call a σ'-matching algorithm σ-complete iff it computes all σ-different solutions to a σ'-matching problem. In our intended

setting, bound variables correspond to abstract meta-variables, while free variables correspond to unresolved existential variables in a matching problem.

3 A Pattern Matching Tactic for Autosubst

We describe a general pattern-matching tactic `as_apply` for terms of the λ-calculus. The tactic is implemented in Rocq's tactic language Ltac.

Let P be an n-ary relation, and consider the following Rocq goal:

H : $\forall$ <qvars>, <premises> $\to P\ t_1\ t_2\ ...\ t_n$
———————————————————————————————
$P\ s_1\ s_2\ ...\ s_n$

with H an assumption in the context we want to apply, where `<qvars>` and `<premises>` denote the quantified variables and premises of H.

A call of `as_apply` H proceeds in two phases: preprocessing and matching.

Preprocessing. We first normalise both the goal and the conclusion of H with respect to the σ-calculus using the `asimpl` tactic provided by Autosubst.

Next, all quantified variables are replaced with existential variables (evars) of the corresponding types, which will be instantiated during matching. The premises of H are then turned into subgoals, to be solved after matching with instantiated quantified variables.

The actual matching will be between the corresponding s_i and t_i. In a first step, we try whether this can be solved by pure syntactic matching using Rocq's tactic. Furthermore, if s_i uses Autosubst's explicit renamings (i.e., is of the form $s\langle\xi\rangle$ or $s[\xi]$) but t_i is not, we change s_i to $s[\xi]$ ($s\langle\xi\rangle$) using `substify` (`renamify`). These are Autosubst-provided tactics that perform conversion between instantiation and renaming operations.

Matching Phase. After preprocessing, we obtain a σ-matching problem

$$\mathcal{M} = \{s_i =_\sigma t_i \mid s_i \text{ and } t_i \text{ are } \sigma\text{-expressions}\}$$

We process equations in $\mathcal{M}$ from left to right.

For each equation $s_i =_\sigma t_i$, we first attempt matching in the $\sigma_{\texttt{min}}$ fragment using a procedure $\sigma_{\texttt{min}}(s_i,\ t_i)$. If successful, this yields a substitution θ, which is applied to all remaining equations t_k for $k \geq i$.

The function $\sigma_{\texttt{min}}(s,\ t)$ is a partial procedure that computes substitutions by matching modulo the $\sigma_{\texttt{min}}$ fragment. It proceeds as follows:

- If s and t match syntactically, a substitution is returned.
- Otherwise, t is rewritten using $\sigma_{\texttt{min}}$ rules (Figure 1), and matching is retried.

In Figure 2, we give an operational description of the $\sigma_{\texttt{min}}$-function. The function performs a backtracking search over possible rewrites of t until a match with s is found. For example, $\sigma_{\texttt{min}}(s[\sigma \circ \tau],\ ?s[\tau])$ matches first with the seventh case, but the execution path stemming from

$$
\begin{aligned}
\sigma_{\texttt{min}}(s,\ s) &\Rightarrow \{\} \\
\sigma_{\texttt{min}}(s,\ ?s) &\Rightarrow \{?s \mapsto s\} \\
\sigma_{\texttt{min}}(\lambda.s,\ \lambda.s') &\Rightarrow \sigma_{\texttt{min}}(s,\ s') \\
\sigma_{\texttt{min}}(s\ t,\ s'\ t') &\Rightarrow \sigma_{\texttt{min}}(t,\ \sigma_{\texttt{min}}(s,\ s')\ t') \cup \sigma_{\texttt{min}}(s,\ s') \\
\sigma_{\texttt{min}}(\sigma \circ \tau,\ \sigma' \circ \tau') &\Rightarrow \sigma_{\texttt{min}}(\tau,\ \sigma_{\texttt{min}}(\sigma,\ \sigma')\ \tau') \cup \sigma_{\texttt{min}}(\sigma,\ \sigma') \\
\sigma_{\texttt{min}}(s \cdot \sigma,\ t \cdot \tau) &\Rightarrow \sigma_{\texttt{min}}(\sigma,\ \sigma_{\texttt{min}}(s,\ t)\ \tau) \cup \sigma_{\texttt{min}}(s,\ t) \\
\sigma_{\texttt{min}}(s[\sigma \circ \tau],\ s'[\sigma']) &\Rightarrow \sigma_{\texttt{min}}(\sigma,\ \sigma_{\texttt{min}}(s,\ s')\ \sigma') \cup \sigma_{\texttt{min}}(s,\ s') \\
\sigma_{\texttt{min}}(\lambda.s[\Uparrow\ \sigma],\ s'[\sigma']) &\Rightarrow \sigma_{\texttt{min}}((\lambda.s)[\sigma],\ s'[\sigma']) \\
\sigma_{\texttt{min}}(s[\sigma]\ t[\sigma],\ s'[\sigma]) &\Rightarrow \sigma_{\texttt{min}}((s\ t)[\sigma],\ s'[\sigma]) \\
\sigma_{\texttt{min}}(s[\sigma \circ \tau],\ s'[\sigma']) &\Rightarrow \sigma_{\texttt{min}}(s[\sigma][\tau],\ s'[\sigma]) \\
\sigma_{\texttt{min}}(\sigma \circ \tau \circ \rho,\ \sigma' \circ \tau') &\Rightarrow \sigma_{\texttt{min}}((\sigma \circ \tau) \circ \rho,\ \sigma' \circ \tau') \\
\sigma_{\texttt{min}}(s[\tau] \cdot \sigma \circ \tau,\ \sigma' \circ \tau') &\Rightarrow \sigma_{\texttt{min}}((s \cdot \sigma) \circ \tau,\ \sigma' \circ \tau')
\end{aligned}
$$

Figure 2: The $\sigma_{\texttt{min}}$-function for matching in the $\sigma_{\texttt{min}}$ fragment

it does not result in a σ-substitution. The execution backtracks and continues by matching with the tenth case, resulting in $\{?s \mapsto s[\sigma]\}$.

If $\sigma_{\texttt{min}}(s,t)$ fails, we apply a small set of heuristics capturing common patterns in developments:

- If $s_i =_\sigma t_i$ is of the form $s[t \cdot \sigma] =_\sigma ?s[t \cdot id]$, instantiate t_k with $\{?s \mapsto s[\Uparrow\ \sigma]\}$, for $k \geq i$.
- If $s_i =_\sigma t_i$ is of the form $s =_\sigma ?s[?\sigma]$, instantiate t_k with $\{?s \mapsto s,\ \sigma \mapsto id\}$ respectively, for $k \geq i$.
- If $s_i =_\sigma t_i$ is of the form $s =_\sigma ?s[id]$, instantiate t_k with $\{?s \mapsto s\}$ respectively, for $k \geq i$.
- If $s_i =_\sigma t_i$ is of the form $s =_\sigma s[?\sigma]$, instantiate t_k with $\{?\sigma \mapsto id\}$ respectively, for $k \geq i$.

We again σ-normalize H with the potentially instantiated existential variables. The tactic succeeds if the goal and H are syntactically equal at this point.

3.1 Generalisation

Autosubst [36] is a code generator that takes a second-order HOAS [28] representation of a custom syntax and generates a model of extended σ-calculus comprising multiple mutually inductive sorts, vector substitutions, and first-class renamings.

This tactic has been generalised to work for the input language of the Autosubst compiler. The $\sigma_{\texttt{min}}$ fragment describes the instantiation operation and has compositionality laws. Hence, for a general syntax, the $\sigma_{\texttt{min}}$-function has congruence cases, and cases corresponding to the instantiation and compositionality laws.

If $\sigma_{\texttt{min}}(s,t)$ fails, we check whether $s =_\sigma t$ matches with certain equations that are recurring in developments. In the case of the first equation $s[t \cdot \sigma] =_\sigma ?s[t \cdot id]$, the expression $s[t \cdot id]$ denotes the elimination of the abstraction $\lambda.s$ by substituting t for the bound index. We generate a case accordingly for each binder in the syntax. For example, if we have a polyadic binder $\lambda_2 : (tm \to\ tm \to tm) \to tm$, we generate a case for the equation $s[t_1 \cdot t_2 \cdot \sigma] = ?s[t_1 \cdot t_2 \cdot id]$. In the

solution $\{?s \mapsto s[\Uparrow\ \sigma]\}$ for the λ-abstraction, the substitution $\Uparrow \sigma$ comes from the instantiation law $(\lambda.s)[\sigma] = \lambda.s[\Uparrow\ \sigma]$. For a general binder, this would be the lifted substitution vector formed during instantiation. In the case of λ_2, the solution would be $\{?s \mapsto s[\Uparrow\Uparrow \sigma]\}$ because $(\lambda_2.s)[\sigma] = \lambda_2.s[\Uparrow\Uparrow\ \sigma]$.

Note that as an additional difficulty since Autosubst supports first-class renamings, the $\sigma_{\texttt{min}}$-function and the specific equations have cases for the renaming operation, which we omit here for conciseness.

3.2 Current Limitations

We summarise some of the limitations of matching in the $\sigma_{\texttt{min}}$ fragment and the `as_apply` tactic.

First note that in itself, the $\sigma_{\texttt{min}}$ fragment is not a confluent rewrite system. The expression $(\lambda.s)[\sigma][\tau]$ reduces to both $\lambda.s[0[\Uparrow\ \tau] \cdot \sigma \circ \uparrow \circ \Uparrow \tau]$ and $\lambda.s[0 \cdot \sigma \circ \tau \circ \uparrow]$ via $\sigma_{\texttt{min}}$ rules. But, they are not joinable as $\sigma_{\texttt{min}}$ lacks the rules $0[s \cdot \sigma] \to s$ and $\uparrow \circ (s \cdot \sigma) \to \sigma$. Of course they are joinable via the whole σ-calculus.

Note that $\sigma_{\texttt{min}}$-matching does not have unitary solutions in general. An example is $s[\sigma \circ \tau] =_{\sigma_{\texttt{min}}} ?s[?\sigma]$ which has two solutions $\{?s \mapsto s,\ ?\sigma \mapsto \sigma \circ \tau\}$ and $\{?s \mapsto s[\sigma],\ ?\sigma \mapsto \tau\}$.

Generally, `as_apply` is not σ-complete. In the above example, it would only produce the first solution. `as_apply` is not complete even as a decider. It fails for the equation $t =_\sigma ?s[t \cdot \sigma]$ though there is the solution $\{?s \mapsto 0\}$. Note that this would rarely be a problem in a practical development.

4 Case Studies

We have tested the tactic on Rocq solutions of the POPLMark [6] and POPLMark Reloaded [2] challenges, two benchmarks on reasoning with binders. In this section, we present examples from our modified solutions. The linked development contains both the original and new solutions.

Example 2. When proving substitutivity of multi-step in the POPLMark Reloaded, we have to prove:

$$
\begin{array}{l}
\mathit{mstep\_inst} : \forall\ s\ t\ \sigma,\ s \gg t \to s[\sigma] \gg t[\sigma] \\
H : \forall n,\ \sigma\ n \gg \tau\ n \\
\hline
(\sigma\ i)\langle\uparrow\rangle \gg (\tau\ i)\langle\uparrow\rangle
\end{array}
$$

The `apply` tactic fails if we try to apply *mstep_inst* because the renaming operation does not match with the instantiation operation, meaning that originally, the renaming operation has to be manually changed to an instantiation. We apply *mstep_inst* with the `as_apply` tactic. After pre-processing, the goal has changed to $(\sigma\ i)[\uparrow] \gg (\tau\ i)[\uparrow]$. The $\sigma_{\texttt{min}}$-function first solves the equation $(\sigma\ i)[\uparrow] = ?s[?\sigma]$ resulting in $\{?s \mapsto \sigma\ i, ?\sigma \mapsto \uparrow\}$, and solves next $(\tau\ i)[\uparrow] = ?t[\uparrow]$ resulting in $\{?t \mapsto \tau\ i\}$.

Example 3. Consider the following case in the context morphism lemma in the POPLMark challenge:

$$
\begin{array}{l}
\mathit{T\_Tapp} : \forall\ \Delta\ \Gamma\ s\ A\ A'\ B,\ \Delta;\Gamma \vdash s : \forall A.B \to \Delta \vdash A' <: A \to \Delta;\Gamma \vdash s\ A' : B[A' \cdot id] \\
H_1 : s[\sigma;\tau] : (\forall A.B)[\sigma] \\
H_2 : \Delta \vdash A'[\sigma] : A[\sigma] \\
\hline
\Delta;\Gamma \vdash s[\sigma;\tau]\ A'[\sigma] : B[A'[\sigma] \cdot \sigma]
\end{array}
$$

The `apply` tactic fails when applying *T_Tapp*, and this is the reason for the originally indirect definition of *T_Tapp*. We apply *T_Tapp* with `as_apply`. After the pre-processing steps, we have to match $\Delta;\Gamma \vdash s[\sigma;\tau]\ A'[\sigma] : B[A'[\sigma]\cdot\sigma]$ with $?\Delta;?\Gamma \vdash ?s\ ?A' : ?B[?A'\cdot id]$. The equation $s[\sigma;\tau]\ A'[\sigma] =?s\ ?A'$ matches syntactically and $\sigma_{\texttt{min}}$-function results in $\{?s \mapsto s[\sigma;\tau], ?A' \mapsto A'[\sigma]\}$. But, $B[A'[\sigma]\cdot\sigma] =?B[A[\sigma]\cdot id]$ can't be matched with $\sigma_{\texttt{min}}$-rules and hence $\sigma_{\texttt{min}}$-function fails. However, it has the form $s[t\cdot\sigma] =?s[t\cdot id]$ described in the last section. Hence, we resolve $?B$ as $B[\Uparrow\sigma]$, and $B[\Uparrow\sigma][A'[\sigma]]$ normalizes to $B[A'[\sigma]\cdot\sigma]$.

Example 4. When proving transitivity of subtyping (POPLMark), we have:

$$\begin{array}{l} \mathit{sub\_weak} : \forall\Delta\ \Delta'\ \xi\ A_1\ A_2,\ \Delta \vdash A_1 <: A_2 \to (\forall x.(\Delta\ x)\langle\xi\rangle = \Delta'(\xi\ x)) \to \Delta' \vdash A_1\langle\xi\rangle <: A_2\langle\xi\rangle \\ H : \Delta \vdash B\langle\uparrow\rangle <: B'\langle\uparrow\rangle \\ \hline (B\langle\uparrow\rangle\cdot\Delta\circ\uparrow) \vdash B\langle\uparrow\rangle <: B'\langle\xi\circ\uparrow\rangle \end{array}$$

We need to apply *sub_weak*, which is the weakening lemma for subtyping. Originally, *sub_weak* has two versions: one with the above statement, and an auxiliary lemma that has the conclusion $\Delta' \vdash A'_1 <: A'_2$ with the extra premises $A'_1 = A_1\langle\xi\rangle$ and $A'_2 = A_2\langle\xi\rangle$ which is applied in this case. We proceed with `as_apply`. All equations match syntactically except $B'\langle\xi\circ\uparrow\rangle =?A_2\langle\uparrow\rangle$. Since $B'\langle\xi\circ\uparrow\rangle$ can be decomposed as $B'\langle\xi\rangle\langle\uparrow\rangle$, the $\sigma_{\texttt{min}}$-function resolves $B'\langle\xi\circ\uparrow\rangle =?A_2\langle\uparrow\rangle$ as $\{?A_2 \mapsto B'\langle\xi\rangle\}$.

Example 5. We look at the proof of *ty_subst* from the POPLMark challenge:

$$\begin{array}{l} \mathit{sub\_subst} : \forall\Delta\ \Delta'\ \sigma\ A\ B,\ \Delta \vdash A <: B \to (\forall x.\Delta' \vdash \sigma\ x <: (\Delta\ x)[\sigma]) \to \Delta' \vdash A[\sigma] <: B[\sigma] \\ H : \Delta \vdash A' <: A \\ \hline \Delta' \vdash A' <: A \end{array}$$

Here, we have a goal in uninstantiated form and an assumption in instantiated form. Originally, A and A' are manually changed to $A[id]$ and $A'[id]$ before applying *sub_subst*. We apply *sub_subst* with `as_apply`. Eventually, we need to match $A' =?A[?\sigma]$ and $A =?B[?\sigma]$. While the $\sigma_{\texttt{min}}$-function fails in both cases, our additional heuristics resolve $?A$, $?B$ and $?\sigma$ as A', A and id respectively.

Discussion. Even in relatively short proof scripts such as the POPLMark challenge (642 lines) and POPLMark Reloaded (683 lines), the `as_apply` is used frequently: 15 times in the POPLMark B and 10 times in the POPLMark Reloaded. In all cases, it successfully solves the intended goal. Compared to the previous solutions, the proposed solutions with the matching tactics avoid the need for indirect definitions, auxiliary lemmas, and the need to manually transform the goal or find the instances for existential variables.

Additional (single) examples, for example, as one in a formalisation of Martin-Löf Type Theory [3], can be found in the appendix. These developments require a customised matching tactic due to manual adaptations of the substitution primitives in the original developments. Apart from these technical problems, both in this case study and in a development of the call-by-push-value [14], we have so far not found any essential problems with the tactic.

5 Related Work

Mechanizing syntax has a long history, leading to many syntax representations [11, 4, 23, 28, 30, 18, 31] and supporting tools [33, 36, 5, 21, 35, 37, 8, 29, 16, 34]. Autosubst [33, 36] provides

automation for de Bruijn syntax and has been used in several mechanizations [2, 14, 3, 13].

Unification in languages with binders has been studied extensively [19]. Traditionally, unification is defined modulo $\beta\eta$-equivalence, known as higher-order unification. While higher-order unification is undecidable in general, Huet's algorithm works well in practice [20]. Miller et al. [24] identify a fragment named *higher-order patterns* for which higher-order unification is decidable and is unitary, i.e. admits general unifiers. Higher-order matching is decidable up to fourth order [27] while for higher orders, decidability remains open.

Calculi for explicit substitutions such as the σ-calculus were introduced by Abadi et al. [1] to bridge the gap between the λ-calculus and its implementations. Curien et al. [10] prove that the σ-calculus is a convergent rewriting system, and later Schäfer et al. [32] show that the σ-calculus is a sound and complete model for the de Bruijn algebra. Together, this means the σ-calculus has practical usage as a rewriting system to decide equality in languages with binders as in the Autosubst library [33, 36]. However, these techniques do not directly extend to matching problems.

Dowek et al. [12] show that higher-order unification is reducible to unification in the first-order equational theory of $\lambda\sigma$-calculus (σ-calculus with β rule), and provide a general unification method for the $\lambda\sigma$-calculus. Moura et al. [25] show that this method does not decide second-order matching in the $\lambda\sigma$-calculus by providing a non-terminating counter-example. They characterise a fragment for which the method terminates and provide a second-order matching algorithm for this fragment. This is not directly applicable as it covers the σ-calculus, including the β rule.

Nominal syntax [15] provides an alternative representation of languages with binders. Urban et al. [38] show that *nominal unification* is both decidable and unitary. Cheney [9] shows that higher-order pattern unification problems can be solved by encoding them as nominal unification problems, and Levy et al. [22] show the reverse. Nantes-Sobrinho et al. [26] generalise nominal unification problems to *nominal equational problems* and provide a rule-based algorithm to find solutions in the ground nominal algebra [15].

Rocq's original unification algorithm relies on heuristics [39]. Ziliani et al. [39] give a new unification algorithm along with an implementation for Calculus of Inductive Constructions, incorporating canonical structures and universe polymorphism. They formally describe a heuristic called *controlled backtracking* used in the unification algorithm of Rocq.

6 Conclusion and Ongoing Work

We extended the Autosubst framework to generate a pattern-matching tactic for custom syntax. The tactic is intended as a substitute for the `apply` tactic when matching with an assumption up to the substitution calculus. We tested this tactic on standard benchmarks, such as the POPLMark [6] and POPLMark Reloaded [2] challenges, and were able to simplify previous solutions.

The current tactic operates heuristically for pragmatic reasons – the σ-matching problem could potentially have many and even infinite solutions. However, in case the solution is not unique, it could produce an incorrect solution when there is a correct solution. Eventually, we are interested in having a matching tactic with proven guarantees; for example, we are interested in identifying the biggest fragment σ' for which the uniqueness criterion holds: If θ is a solution of $s =_{\sigma'} t$, then any other solution θ' is σ-equal to θ.

References

[1] Martin Abadi, Luca Cardelli, P-L Curien & J-J Lévy (1989): *Explicit substitutions*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 31–46.

[2] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark reloaded: Mechanizing proofs by logical relations*. *Journal of Functional Programming* 29, p. e19.

[3] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot & Loïc Pujet (2024): *Martin-Löf à la Coq*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 230–245.

[4] Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack & Stephanie Weirich (2008): *Engineering formal metatheory*. *Acm sigplan notices* 43(1), pp. 3–15.

[5] Brian Aydemir & Stephanie Weirich (2010): *LNgen: Tool support for locally nameless representations*.

[6] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized metatheory for the masses: the POPLMark challenge*. In: *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005. Proceedings 18*, Springer, pp. 50–65.

[7] Franz Baader & Tobias Nipkow (1998): *Term rewriting and all that*. Cambridge university press.

[8] Jan van Brügge, Andrei Popescu & Dmitriy Traytel (2025): *Animating MRBNFs: Truly modular binding-aware datatypes in Isabelle/HOL*. In: *16th International Conference on Interactive Theorem Proving (ITP 2025)*, 352, Sheffield.

[9] James Cheney (2005): *Relating nominal and higher-order pattern unification*. In: *Proceedings of the 19th international workshop on Unification (UNIF 2005)*, LORIA research report A05, pp. 104–119.

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

[11] Nicolaas Govert De Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. In: *Indagationes mathematicae (proceedings)*, 75, Elsevier, pp. 381–392.

[12] Gilles Dowek, Thérese Hardin & Claude Kirchner (1995): *Higher-order unification via explicit substitutions*. In: *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, IEEE, pp. 366–374.

[13] Yannick Forster, Dominik Kirst & Dominik Wehr (2021): *Completeness theorems for first-order logic analysed in constructive type theory: Extended version*. *Journal of Logic and Computation* 31(1), pp. 112–151.

[14] Yannick Forster, Steven Schäfer, Simon Spies & Kathrin Stark (2019): *Call-by-push-value in Coq: operational, equational, and denotational theory*. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 118–131.

[15] Murdoch J Gabbay & Andrew M Pitts (2002): *A new approach to abstract syntax with variable binding*. *Formal aspects of computing* 13(3), pp. 341–363.

[16] Andrew Gacek (2008): *The Abella interactive theorem prover (system description)*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 154–161.

[17] Mathews George: *Extended Autosubst compiler, Case studies and Work-in-Progress developments*. `https://github.com/mthwsgrg/lfmtp2026`. (visited on May 1 2026).

[18] Andrew D Gordon (1993): *A mechanisation of name-carrying syntax up to alpha-conversion*. In: *HOL Users' Group Workshop*, Springer, pp. 413–425.

[19] Jean Goubault-Larrecq & Gopalan Nadathur (2001): *Higher-order unification and matching. Handbook of automated reasoning* 2, p. 1009.

[20] Gerard P. Huet (1975): *A unification algorithm for typed λ-calculus. Theoretical Computer Science* 1(1), pp. 27–57.

[21] Steven Keuchel, Stephanie Weirich & Tom Schrijvers (2016): *Needle & Knot: Binder boilerplate tied up*. In: *European Symposium on Programming*, Springer, pp. 419–445.

[22] Jordi Levy & Mateu Villaret (2008): *Nominal unification from a higher-order perspective*. In: *International Conference on Rewriting Techniques and Applications*, Springer, pp. 246–260.

[23] Conor McBride & James McKinna (2004): *Functional pearl: i am not a number–i am a free variable*. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pp. 1–9.

[24] Dale Miller (1991): *A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of logic and computation* 1(4), pp. 497–536.

[25] Flávio LC de Moura, Fairouz Kamareddine & Mauricio Ayala-Rincón (2005): *Second-order matching via explicit substitutions*. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, pp. 433–448.

[26] Daniele Nantes-Sobrinho, Maribel Fernandez, Deivid Vale & Mauricio Ayala-Rincón (2025): *A Nominal Approach to Equational Problems in Languages with Binders. ACM Transactions on Computational Logic* 27(1), pp. 1–46.

[27] Vincent Padovani (2000): *Decidability of fourth-order matching. Mathematical Structures in Computer Science* 10(3), pp. 361–372.

[28] Frank Pfenning & Conal Elliott (1988): *Higher-order abstract syntax. ACM sigplan notices* 23(7), pp. 199–208.

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

[30] Andrew M Pitts (2001): *Nominal logic: A first order theory of names and binding*. In: *International Symposium on Theoretical Aspects of Computer Software*, Springer, pp. 219–242.

[31] Piotr Polesiuk & Filip Sieczkowski (2024): *Functorial Syntax for All*.

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

[33] Steven Schäfer, Tobias Tebbi & 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, Proceedings 6*, Springer, pp. 359–374.

[34] Carsten Schürmann (2009): *The Twelf proof assistant*. In: *International Conference on Theorem Proving in Higher Order Logics*, Springer, pp. 79–83.

[35] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar et al. (2010): *Ott: Effective tool support for the working semanticist. Journal of functional programming* 20(1), pp. 71–122.

[36] Kathrin Stark, Steven Schäfer & 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*, pp. 166–180.

[37] Christian Urban & Cezary Kaliszyk (2012): *General bindings and alpha-equivalence in Nominal Isabelle. Logical methods in computer science* 8.

[38] Christian Urban, Andrew M Pitts & Murdoch J Gabbay (2004): *Nominal unification. Theoretical Computer Science* 323(1-3), pp. 473–497.

[39] Beta Ziliani & Matthieu Sozeau (2015): *A unification algorithm for Coq featuring universe polymorphism and overloading.* In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pp. 179–191.

A More Examples

We look at more examples from the MLTT mechanization [3][1], POPLMark Reloaded, and POPLMark to demonstrate the usefulness of our tactic.

Example 6. In the MLTT mechanization, in the proof of *complete_Pi*, we come across the following.

$$ty\_app\ :\ \forall\ \Gamma\ (f\ d\ dom\ cod : tm),\ \Gamma \vdash f : \Pi_{dom}\ cod \to \Gamma \vdash d : dom \to \Gamma \vdash tApp\ f\ d : cod[d \cdot id]$$
$$H\ :\ \Gamma \vdash a : dom\langle\xi\rangle$$
$$\Gamma \vdash tApp\ n\langle\xi\rangle\ a\ :\ cod[a \cdot \xi]$$

Here, we need to apply the type application rule *ty_app* for the Π type. The `apply` tactic would fail, and we resort to `as_apply`. First, we do the pre-processing steps, and all the equations are matched except $cod[a \cdot \xi] =?cod[a \cdot id]$. This equation can't be matched in the $\sigma_{\texttt{min}}$ fragment. But, it's of the form $s[t \cdot \sigma] =?s[t \cdot id]$. Since ξ is a renaming, we resolve $?cod$ as $cod\langle \Uparrow \xi\rangle$. After, $cod\langle \Uparrow\ \xi\rangle[a \cdot id]$ is reduced to $cod[a \cdot \xi]$.

Example 7. This example is a continuation from the above example, where we have to prove the sub-goal (first premise of *ty_app*) by applying the weakening lemma *ty_wk*.

$$ty\_wk\ :\ \forall\ \Gamma\Delta,\ \vdash \Delta \to \Gamma \vdash t : A \to \Delta \vdash t\langle\xi\rangle : A\langle\xi\rangle$$
$$\Gamma \vdash n\langle\xi\rangle : \Pi_{dom\langle\xi\rangle}\ cod\langle \Uparrow \xi\rangle$$

The `apply` tactic fails as expected. We apply the `as_apply` tactic on *ty_wk* and eventually we face the σ-matching equation $\Pi_{dom\langle\xi\rangle}\ cod\langle \Uparrow \xi\rangle =?A\langle\xi\rangle$. This equation matches in the $\sigma_{\texttt{min}}$ fragment because we have the instantiation law for renaming $(\Pi_{dom}\ cod)\langle\xi\rangle = \Pi_{dom\langle\xi\rangle}\ cod\langle \Uparrow \xi\rangle$. Hence, the $\sigma_{\texttt{min}}$-function resolves $?A \mapsto \Pi_{dom}\ cod$.

Example 8. In the *step_inst* lemma from the POPLMark Reloaded, we face the following.

$$\beta\ :\ \forall s\ t,\ (\lambda.s)\ t \succ s[t \cdot id]$$
$$\lambda.s[\Uparrow \sigma]\ t[\sigma] \succ s[t[\sigma] \cdot \sigma]$$

The `apply` tactic can syntactically match the reducible expression $(\lambda.s[\Uparrow\ \sigma])\ t[\sigma]$ with $(\lambda.s)\ t$ in the β-rule. But `apply` fails to match $s[t[\sigma] \cdot \sigma]$ with $s[\Uparrow\ \sigma][t[\sigma] \cdot id]$ because it relies on the Rocq unification engine that doesn't convert up to σ-rules. In the case of `as_apply`, we resolve the variables $?s$ and $?t$ as $s[\Uparrow\ \sigma]$ and $t[\sigma]$ respectively. Finally, we do a σ-normalization that reduces $s[\Uparrow\ \sigma][t[\sigma] \cdot id]$ to $s[t[\sigma] \cdot \sigma]$.

Example 9. We look at the proof of the context morphism lemma from the POPLMark. At one point in the proof, we have the context and goal as follows:

$$context\_renaming\_lemma\ :\ \forall \Delta'\ \Delta\ \Gamma\ \Gamma'\ s\ A\ \xi\ \zeta\ \ldots\ \to \Delta';\Gamma' \vdash s : A \to \Delta;\Gamma \vdash s\langle\xi;\zeta\rangle : A\langle\xi\rangle$$
$$\Delta'; A[\sigma] \cdot \Gamma' \vdash\ (\tau\ f)\langle id;\uparrow\rangle : (\Gamma\ f)[\sigma]$$

The `apply` tactic fails when we try to apply the context renaming lemma. The `as_apply` proceeds as expected. The equation $(\tau\ f)\langle id;\uparrow\rangle =?s\langle?\xi;?\zeta\rangle$ matches syntactically and we resolve $\{?s \mapsto \tau\ f, ?\xi \mapsto id, ?\zeta \mapsto \uparrow\}$. Now, the $\sigma_{\texttt{min}}$-function fails for the equation $(\Gamma\ f)[\sigma] =?A\langle id\rangle$. But it has the form $s =?t\langle id\rangle$ (for renamings). Hence, we resolve $?A$ as $(\Gamma\ f)[\sigma]$

[1] https://github.com/CoqHott/logrel-coq/blob/coq-8.19/theories/LogicalRelation/Neutral.v