Coq à la Carte
A Practical Approach to Modular Syntax with Binders

Yannick Forster
Saarland University
Saarland Informatics Campus, Saarbrücken, Germany
forster@ps.uni-saarland.de

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

Abstract
The mechanisation of the meta-theory of programming languages is still considered hard and requires considerable effort. When formalising properties of the extension of a language, one hence wants to reuse definitions and proofs. But type-theoretic proof assistants use inductive types and predicates to formalise syntax and type systems, and these definitions are closed to extensions. Available approaches for modular syntax are either inapplicable to type theory or add a layer of indirectness by requiring complicated encodings of types.

We present a concise, transparent, and accessible approach to modular syntax with binders by adapting Swierstra’s Data Types à la Carte approach to the Coq proof assistant in an ad-hoc way. Our approach relies on two phases of code generation: We extend the Autosubst 2 tool and allow users to specify modular syntax with binders in a HOAS-like input language. To state and automatically compose modular functions and lemmas, we implement commands based on MetaCoq. We support modular syntax, functions, predicates, and theorems.

We demonstrate the practicality of our approach by modular proofs of preservation, weak head normalisation, and strong normalisation for several variants of mini-ML.

1 Introduction
Despite all efforts, 15 years after the POPLMark challenge [3], mechanising proofs concerning syntax with binders in proof assistants is still considered hard. Besides the treatment of binders, both the POPLMark and the POPLMark Reloaded [2] challenge hence focus attention on component reuse. Component reuse covers both reusing definitions and parts of proofs. However, to the best of our knowledge, all submitted solutions to either challenge follow a copy-paste approach and do not actually reuse proofs.

Copy-pasting proofs results in inelegant and hard-to-maintain developments, but so far, there is hardly an alternative. While suggestions how to use modular syntax [5, 10, 17, 18, 24] for proof assistants like Coq and Agda exist, we failed to locate a development based on one of the proposed solutions, apart from the case studies contained in the publications. The POPLMark challenge suggests three evaluation criteria to judge the practicality of a formalisation: conciseness, transparency, and accessibility. These criteria are directly applicable to evaluate the practicality of an approach for modular syntax: the overhead in using the modular approach should be reasonable (conciseness), the content of definitions and theorems should be apparent to someone unfamiliar with the approach (transparency), and the cost of entry should be reasonable (accessibility).

In the programming context, the problem of reusing definitions is called the expression problem [31]:

“The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code.”

One can take this as a fourth evaluation criterion: true modularity, i.e. proof terms should be checked only once.

The Data Types à la Carte approach of Swierstra [29] proposes a solution in Haskell, where expressions are defined as a supertype parameterised by a functor F which is used to instantiate the type with so-called features. For example, arithmetic, boolean, and lambda features are encoded as:

data Exp F = In (F (Exp F))
Arith X = (X, X) + N       -- addition and nat. constants
Booleans X = (X, X, X) + B   -- if and boolean constants
Lambda X = N + (X, X) + X   -- variables, app., abstraction

Several features are dynamically combined via coproducts of functors, while functions can be defined as algebras with the help of type classes. Although the development makes heavy use of type classes, it fulfils the criteria of being concise, transparent, accessible, and truly modular.

Unfortunately, the definition of Exp via an arbitrary functor is impossible in proof assistants as Coq or Agda, which require defined types to be strictly positive [8].

To the best of our knowledge, all adaptions of Data Types à la Carte to proof assistants hence add a layer of indirectness to circumvent this problem: Schwaab and Siek [24] formalise the syntax of (some) strictly positive functors and only allow such instances; other approaches work with Church encodings [10] or containers and proof algebras [17].

While elegant in theory, these approaches fail the evaluation criteria of the POPLMark challenge: All approaches
require many lines of preliminary code, lacking conciseness. To understand the statement of theorems, a deep understanding of the encoding is necessary, lacking transparency. Finally, a user has to learn about the encoding, for example, how to inductive data types using containers, lacking accessibility. Further, if we work via a theory of codes, Coq's internal support to define inductive types, do proofs by induction, or define functions by recursion can not be used at all.

As a consequence, users of proof assistants do not employ modular syntax in practice. The goal of this paper is to present a practical approach to modular syntax.

Practicality comes at a cost, and in our case as an assumption: We claim that, in practice, we do not need a dynamically extensible type \( \text{Exp} \) that can be instantiated to \( \text{Exp} F \) with arbitrary combinations \( F \) of features; fixed, static instantiations \( \text{Exp}_{F_1}, \ldots, \text{Exp}_{F_n} \) suffice to define modular functions, inductive modular predicates over modular syntax, and modular proofs over these definitions. This ad-hoc approach already yields concise, transparent, accessible, and truly modular developments in Coq.

In a second step, we increase the usability of this approach and automatically generate these instantiations. We extend the Autosubst 2 tool [28] to allow modular specifications of syntax. The user chooses which set of features \( F_i \) should be present in an instantiation, and a type \( \text{Exp}_{F_i} \) is generated for which definitions and lemmas given modularly before can be composed automatically.

The code generation even improves on Data Types à la Carte: The user does not have to write any preliminary code to work with modular syntax. We use the meta-programming facilities of MetaCoq [26] to program both input and composition mechanisms for modular functions and lemmas.

We explain our approach for syntax, functions on this syntax, and proofs on both (Section 2), for induction principles (Section 3), and for (modular) inductive predicates over modular syntax (Section 4). In Section 5, we elaborate on implementation details, mainly on our extension to Autosubst 2 and our use of MetaCoq. We then showcase our approach on a wealth of case studies handling inductive definitions, inductive predicates, functions and proofs over these definitions in Sections 6 and 7. Precisely, we mechanise:

- Type preservation for a language with natural numbers, arrays and options, which is the case study used in [24]. Our development seems to be similar in size.
- Monotonicity and type preservation of big-step evaluation for mini-ML (i.e. the simply-typed \( \lambda \)-calculus with natural numbers, arithmetic, and recursive abstractions, often also called PCF), which is the case study used in [10, 17]. Our development needs about 625 lines, compared to about 5250 and 5500 lines. See Section 7 for a detailed comparison.
- Type preservation of small-step reduction in the simply-typed \( \lambda \)-calculus with natural numbers and booleans.
- Weak and strong normalisation of small-step reduction in the simply-typed \( \lambda \)-calculus with natural numbers and booleans, inspired by one of the case studies posed as part of the POPLMark Reloaded challenge [2].

**Contributions.** This paper revisits Data Types à la Carte [29] and makes it directly usable in the Coq proof assistant. To ease using modular syntax, we implement code generation as an extension of Autosubst 2 [28] to support arbitrary, but fixed combinations of features. We handle modular inductive types, modular inductive predicates over modular types, and modular functions and proofs over both.

We offer tool support for the definition and composition of such modular components both with and without binders based on MetaCoq [26]. We present several extensive case studies, including the first truly modular mechanised proof of strong normalisation for the simply-typed \( \lambda \)-calculus, and a detailed comparison with related work.

Note that while our developed tools ease working with modular syntax considerably, the approach is also feasible to use with no tool support.

## 2 Modular Syntax

In this section, we give a high-level overview of modular syntax via our ad-hoc adaptation of the Data Types à la Carte approach [29] to Coq. For this introduction, we do not use our developed tools, but highlight code that can be generated automatically with a grey background. At the end of each part, we elaborate on how our tools ease working with modular syntax with binders even more.

For this example, we use the following definition of \( \lambda \)-expressions using de-Bruijn indices:

\[
\begin{align*}
  s, t : \text{exp}_1 & ::= \text{var} \mid \text{app} \; s \; t \; \lambda \; s \\
  x & \in \mathbb{N}
\end{align*}
\]

Imagine to be in a situation where we prove several theorems about this calculus, for example, preservation and normalisation for a simple type system. Later, we decide that we want to extend expressions with booleans and natural numbers, for example, to obtain two distinct calculi with these features. On paper, we would define the extensions as follows:

\[
\begin{align*}
  s, t, u : \text{exp}_2 & ::= \ldots \mid \text{b} \; \text{if} \; s \; \text{then} \; t \; \text{else} \; u \\
  s, t : \text{exp}_3 & ::= \ldots \mid n \; s + t
\end{align*}
\]

To prove preservation and normalisation on paper, we would only explain the cases of proofs concerning the new constructors and refer to the old proofs for the other cases. In this paper, we mirror this situation in Coq and allow a user to define syntax modularly for mechanised proofs.
2.1 Modular Inductive Data Types

The standard Data Types à la Carte approach [29] defines extensible expressions as

\[
\text{Inductive Exp (F : Type → Type) :=}
\begin{align*}
\text{In} & : F (\text{Exp F}) \rightarrow \text{Exp F}. \\
\text{begin} & : \text{Exp F} \rightarrow \text{Exp F}. \\
\end{align*}
\]

Together with feature functors, for our example defined in Figure 1. As a convention, specific feature functors always have a symbol as a subscript, e.g. \(\text{exp}\).

With these definitions and the pointwise coproduct on functors, written \(\bigoplus\), we can define different instantiations of \(\text{exp}\) as:

\[
\begin{align*}
\text{Definition exp}_1 & := \text{Exp} (\text{exp}_1 \bigoplus \text{exp}_\text{var}). \\
\text{Definition exp}_2 & := \text{Exp} (\text{exp}_2 \bigoplus \text{exp}_\text{var} \bigoplus \text{exp}_\text{exp}). \\
\text{Definition exp}_3 & := \text{Exp} (\text{exp}_3 \bigoplus \text{exp}_\text{var} \bigoplus \text{exp}_\text{exp}). \\
\end{align*}
\]

However, Coq’s positivity checker rejects the definition of \(\text{Exp}\), because it would introduce a logical inconsistency via the negation functor \(F(T) := T \rightarrow \bot\).

Instead of a generalised fixed-point type \(\text{Exp : (Type → Type) → Type}\), our ad-hoc approach simply inlines the above definitions, see Figure 2. We write \(\text{exp}_1\), \(\text{exp}_2\), and \(\text{exp}_3\) for different choices of the type \(\text{exp}\); in our later proof development, we simply use separate files.

**Tool support.** Our extension of Autosubst 2 automatically generates the definitions in Figures 1 and 2. Autosubst supports a HOAS [19] specification language where syntax and features can be defined. Negative occurrences of types (here the first \(\text{exp}\) in the type of \(\text{ab}\)) are translated to binders in the output. To generate the above types, one would use the HOAS signature depicted in Figure 3. Each feature is surrounded by a \texttt{begin... end} block, each instantiation can be generated via the \texttt{compose} command.

---

\(\text{exp}_1\)

\[
\begin{align*}
\text{Inductive exp}_1 & := \\
\text{inj}_{\text{var}} & : \text{exp}_{\text{var}} \rightarrow \text{exp}_{\text{var}} \rightarrow \text{exp}_1, \\
\text{inj}_{\lambda} & : \text{exp}_1 \rightarrow \text{exp}_1 \rightarrow \text{exp}_1. \\
\end{align*}
\]

\(\text{exp}_2\)

\[
\begin{align*}
\text{Inductive exp}_2 & := \\
\text{inj}_{\text{var}} & : \text{exp}_{\text{var}} \rightarrow \text{exp}_2, \\
\text{inj}_{\lambda} & : \text{exp}_1 \rightarrow \text{exp}_2, \\
\text{inj}_{\text{constNat}} & : \text{exp}_0 \rightarrow \text{exp}_2. \\
\end{align*}
\]

\(\text{exp}_3\)

\[
\begin{align*}
\text{Inductive exp}_3 & := \\
\text{inj}_{\text{var}} & : \text{exp}_{\text{var}} \rightarrow \text{exp}_3, \\
\text{inj}_{\lambda} & : \text{exp}_1 \rightarrow \text{exp}_3, \\
\text{inj}_{\text{constNat}} & : \text{exp}_0 \rightarrow \text{exp}_3. \\
\end{align*}
\]
We further develop proofs over modular syntax; which are, and then show the statement for the separate features: show that every expressions has leaves, i.e.\[ \forall e : \exp_\var \Rightarrow |e|_\var \geq 0. \]

In a second step, the counting function for \( \exp_\var \) can be obtained, as shown in Figure 5: By a simple case analysis calling the respective feature functions. Since e.g. \( |.|_\var \) only uses \( |.| \) on structurally smaller arguments, this definition is terminating.

**Tool support.** We provide syntax for both the definition and combination of modular fixpoints. Instead of the definition of \( |.|_\var \) in Figure 4, a user can write:

**MetaCoq Run**

```
Modular Fixpoint |.|_\var where \exp_\var \exp extends exp with |.| :=
  fun (s : \exp_\var \exp) \Rightarrow
  match s with
  | abs s \Rightarrow |s|
  | app s t \Rightarrow |s| + |t|
end.
```

And instead of the code in Figure 5, a user can write:

**MetaCoq Run Compose**

```
Fixpoint |.| (e : \exp_\var) : \N :=
match e with
| \inj_\var e \Rightarrow |e|_\var
| \inj_\lambda e \Rightarrow |e|_\lambda
| \inj_\beta e \Rightarrow |e|_\beta
end.
```

Figure 5. Definition of counting function.

All proofs are by an easy case analysis on \( e \). For example, in the application case, we have to prove that

\[ |\app s t|_\beta = |s| + |t| > 0 \]

where \(|s|\) and \(|t|\) are larger than 0 by the assumption \( \text{count}_{\var} \), and so the whole claim follows.

The lemma for e.g. the instantiation \( \exp_\var \) now follows immediately from the respective lemmas for \( \exp_\lambda \) and \( \exp_\beta \):

```
Fixpoint count_{\var} (e : \exp_\var) : |s| > 0.
Proof.
  destruct e; cbn;
  [ apply count_{\var}_\var | apply count_{\var}_\lambda | apply count_{\var}_\beta ];
eauto.
Qed.
```

Since Coq’s induction principle for \( \exp_\var \) is too weak, we do the proof by direct recursion\(^6\) on the expression \( e \) rather than induction. We fix this deficiency in Section 3 and introduce modular induction principles.

**Tool support.** Since assuming \( \text{count}_{\var} \) with variables manually duplicates code, we implement a Coq command to define modular lemmas as follows:

**MetaCoq Run**

```
Modular Lemma count_{\var}_\var where \exp_\var \exp extends exp at 0 with \[ |.|_\var : \forall s, |s|_\var > 0. \]
```

Stating this is equivalent to the following two commands:

Variable count_{\var} : \forall s, |s| > 0.

Lemma count_{\var}_\var : \forall s, |s|_\var > 0.

To combine lemmas, a user can write:

**MetaCoq Run Compose**

```
Lemma count_{\var} on 0 : \forall (s : \exp_\var), |s| > 0.
```

### 2.4 Modular Constructors

We want to lift the constructors from features, e.g. \( \app \) to constructors for an instantiation, e.g. \( \exp_\var \). Smart constructors [29] combine the constructors of the modular type \( \exp_\\beta \) with the actual constructors of \( \exp_\var \), i.e.

```
Definition \app_\\beta st := \inj_\\beta (\app st).
```

However, more instantiations like \( \exp_\\var \) again lead to code duplication. We mirror Swierstra with tight retracts between types, defined in Coq using type classes [27] as follows:

**Class**

```
\class X <: Y :=
  \{ inj : X \rightarrow Y; retr : Y \rightarrow \text{option} X; retract_works : \forall x, \text{retr} (\text{inj} x) = \text{Some} x; retract_tight : \forall x y, \text{retr} y = \text{Some} x \rightarrow \text{inj} x = y \}.
```

The function \( \text{inj} \) of a retract is injective, i.e. if \( \text{inj} x = \text{inj} y \), then also \( x = y \). We can easily define the following instance of the type class:

\(^6\)In the Coq proofs, this requires that \( \text{count}_{\var}_\lambda \) is closed via the Defined and not the Qed keyword.
Using the retract typeclass, we define a more general version of constructors, e.g.:

**Definition** `app < (exp) (exp < exp) s t := inj (app s t)`.

Similarly, we define constructors `if < _then _else _` and `constBool <` to use in arbitrary contexts:

```coq
Check (app < (if < (constBool < true) then var < 1 else var < 2)) t).
```

### Code generation

Autosubst automatically generates the proofs for `exp_retract<1>`, `exp_retract<2>`, and the definition of smart constructors, e.g. `app<>, var<>, and if<>_then _else _`.

#### 3.5 Introduction of a New Feature

If we extend our definitions to the type `exp<var>`, we have to define `|n|` and prove that it returns numbers greater than 0.

For this, we need a new section. We directly use the most concise syntax using our custom commands:

```coq
Section Arith.
Variable exp : Type.
MetaCoq Run Modular
Fixpoint |n| where (exp<n> exp) extends exp with |n| :=
  fun (s : exp<n> exp) =>
    match s with
    | constNat _ => 1
    | plus s t => 1 + |s| s + |t|
  end.
MetaCoq Run Modular
Lemma count_gt where (exp<n> exp) extends exp
  with |n| : ∀ s, |s|>0.
Next Obligation. (* ... *) Defined.
End Arith.
MetaCoq Run Compose Fixpoint |n| on 0 : ∀ (s : exp<n>), N.
MetaCoq Run Compose Lemma count_gt on 0 : ∀ (s : exp<n>), |s|>0.
```

W.r.t. the examples, the ad-hoc definitions offer the same power as a dynamically extensible type of Data Types à la Carte [29]. We essentially defined simple modular data types, modular functions over them and extended the approach to proofs. Our code generation supports the automatic definition of feature functors and combined types bases on a HOAS input language, and we provide commands to define and combine modular functions and lemmas directly.

### 3 Modular Induction Principles

The induction principle Coq generates for e.g. the type `exp<2>` reads as follows:

```coq
Theorem exp<2>_ind : 
∀ P : exp<2> → Prop,
```

Using the modular induction principle, we can obtain an alternative modular proof that every expression has leaves by proving the following lemma first, which can now be defined opaquely using Qed in Coq:

**Lemma 3.2**

1. If `|e' |<v| e. |e' |> 0, then |e|<v| > 0.
2. If `|e' |<1| e. |e' |> 0, then |e|<1| > 0.
3. If `|e' |<e| e. |e' |> 0, then |e|<e| > 0.

**Proof.** By the induction principle from Theorem 3.1 and Lemma 3.2.

### 4 Modular Dependent Predicates

We extend our approach to modular inductive predicates with dependent types over modular syntax. We first define a type system and reduction relation for terms in `exp<n>` and `exp<2>` and extend it to `exp<2>` and `exp<1>` in Section 6.

As types, we use natural numbers and boolean:
As a side-effect of this, we use smart constructors everywhere (as long as there is a proof): we use the modular versions.

\[ \Gamma \vdash s : N \quad \Gamma \vdash t : N \]

\[ \Gamma \vdash \text{atom}_< n : N \quad \Gamma \vdash s +_c t : N \]

\[ \Gamma \vdash \text{boolConst}_< b : \text{TBool} \]

\[ \Gamma \vdash s : \text{TBool} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A \]

\[ \Gamma \vdash \text{if}_< b \text{ then } e_1 \text{ else } e_2 : A \]

\[ \text{Figure 7. Typing for arithmetic and boolean expressions.} \]

\[ s >_s s' \quad t >_t t' \quad s +_c t >_s s' +_c t' \]

\[ \text{atom}_< m +_c \text{atom}_< n >_s \text{atom}_< (m + n) \]

\[ \text{if}_< \text{constBool}_< \text{true} \text{ then } e_1 \text{ else } e_2 >_s e_1 \]

\[ \text{if}_< \text{constBool}_< \text{false} \text{ then } e_1 \text{ else } e_2 >_s e_2 \]

\[ e_1 > e'_1 \quad \text{if}_< e_1 \text{ then } e_2 \text{ else } e_3 >_s \text{ if}_< e'_1 \text{ then } e_2 \text{ else } e_3 \]

\[ e_1 > e'_1 \quad \text{if}_< e_1 \text{ then } e_2 \text{ else } e_3 >_s \text{ if}_< e_1 \text{ then } e'_2 \text{ else } e_3 \]

\[ e_1 > e'_1 \quad \text{if}_< e_1 \text{ then } e_2 \text{ else } e_3 >_s \text{ if}_< e_1 \text{ then } e_2 \text{ else } e'_3 \]

\[ \text{Figure 8. Reduction for arithmetic and boolean expressions.} \]

\[ \Gamma \vdash s : A \quad \Gamma \vdash s : A \quad s >_s s' \quad s >_s s' \]

\[ \Gamma \vdash s : A \quad \Gamma \vdash s : A \quad s >_s s' \quad s >_s s' \]

\[ \text{Figure 9. Typing and reduction for full expressions.} \]

We define modular typing relations \( \vdash_N \) and \( \vdash_B \):

\[ \text{Inductive } \tau_N := \tau_{\text{TNat}}. \]

\[ \text{Inductive } \tau_B := \tau_{\text{TBool}}. \]

\[ \text{list } \tau \rightarrow \tau_N \rightarrow \tau \rightarrow \tau_B \rightarrow \tau_B \rightarrow \text{Prop} \]

\[ \text{As before, the relations assume a type } \tau \text{ and relations } \vdash_N : \vdash_B : \vdash_N : \vdash_B : \text{Prop} \]

\[ \text{As before, the relations assume a type } \tau \text{ and relations } \vdash_N : \vdash_B : \vdash_N : \vdash_B : \text{Prop} \]

\[ \text{We also have to assume retract rules as before, e.g. } \exp_N \exp <_s \exp. \]

\[ \text{As a side-effect of this, we use smart constructors everywhere.} \]

\[ \text{Similar to the assumption of retracts between types, we have to assume that the modular versions of predicates can be embedded into the full predicates. We do not require inclusions, because for predicates the proof itself is irrelevant (as long as there is a proof):} \]

\[ \Gamma \vdash_i s : A \rightarrow \Gamma \vdash s : A \]

\[ s >_i t \rightarrow s > t \]

\[ \text{To show preservation, we have to invert typing rules. We thus assume that the predicate } \vdash_i \text{ agrees with } \tau_i \text{ on terms of the form } \text{inj}_i s : \]

\[ \Gamma \vdash \text{inj}_i s : A \rightarrow \Gamma \vdash \text{inj}_i s : A \]

\[ \text{We now give a modular proof of type preservation for this language. We want to show that if } \Gamma \vdash s : A \text{ and } s >_t t, \text{ then } \Gamma \vdash t : A, \text{ by induction on } s > t. \text{ Similar to before, we use the modular versions } \tau_i \text{ in the modular statements for arguments we want to do induction on. Otherwise, we always use the full versions. The rest of the proof is then very similar to the proofs in the last section:} \]

\[ \text{Lemma 4.1} \quad \text{Assume that if } \Gamma \vdash s : A \text{ and } s > t, \text{ then } \Gamma \vdash t : A. \]

1. If \( \Gamma \vdash s : A \) and \( s >_s t \), then \( \Gamma \vdash t : A \).

2. If \( \Gamma \vdash s : A \) and \( s >_B t \), then \( \Gamma \vdash t : A \).

\[ \text{Proof. We show the claim for arithmetic expressions by case analysis on } s >_s t. \text{ As } \Gamma \vdash_N s : A \text{ via eq. (6), we can do an inversion on the derivation. For } \Gamma \vdash t : A \text{ it suffices to show that } \Gamma \vdash_N t : A \text{ via eq. (4)} \text{ and so the claim holds.} \]

\[ \text{It is again easy to deduce preservation for the combined types, defined as follows:} \]

\[ \text{Inductive } \exp := \text{inj}_N : \exp_N \exp \rightarrow \exp | \text{inj}_B : \exp_B \exp \rightarrow \exp. \]

\[ \text{Inductive } \tau := \text{inj}_N : \tau_N \rightarrow \tau N \rightarrow \tau | \text{inj}_B : \tau_B \rightarrow \tau B \rightarrow \tau. \]

\[ \text{The relations for combined types are in Figure 9.} \]

\[ \text{Theorem 4.2} \quad \text{If } \Gamma \vdash s : A \text{ and } s > t, \text{ then } \Gamma \vdash t : A. \]

\[ \text{Proof. By induction on } s > t \text{ and Lemma 4.1.} \]

\[ \text{5 Tool Support for Modular Syntax} \]

To make modular syntax more convenient to use, we implement three kinds of tool support for modular syntax.

First, we extend the HOAS-like input language of Autosubst 2 [28] to support modular types. Based on this input, we implement static code-generation of feature functors and combined types together with retracts, smart constructors, and modular induction principles.

Second, we extend the automation of Autosubst to support modular syntax. A user can then use instantiation and the asimp1 tactic simplifying substitution goals also on modular syntax.

And third, we implement dynamic code generation based on MetaCoq [26] to ease the statement of modular fixpoints and lemmas and fully automate the composition of this fixpoint and lemmas.

This section can be seen as a limited reference manual; we refer to the case studies both in the next section and in the Coq code for examples.
5.1 Static Code Generation for Modular Syntax

We extend Autosubst’s [28] interface to modular types. Recall Figure 3 for an example input. Autosubst generates functors, types, rejections, smart constructors and induction principles based on this input.

Definition of Functors. For each feature $F$, every type $T$ in $F$ with constructors $C_1, \ldots, C_n$, Autosubst generates a functor $T.F$ with constructors $C_1, \ldots, C_n$.

Definition of Inductive Types. For each specified instantiation $I$ and all types $T_1, \ldots, T_m$ defined in a feature $F$ of $I$, Autosubst generates the types $T_1, \ldots, T_m$ combining all specified features in a file $I$. The constructors of $T$ are called $\text{inj}_I.T.F$.

Retracts. For each specified instantiation $I$ and all types $T_1, \ldots, T_m$ defined in a feature $f$ of $I$, Autosubst proves $T.T.F < T$.

Smart Constructors. For each constructor $C$, Autosubst automatically defines the respective smart constructor called $\text{C}$ via injections.

Induction Principles. For every feature $F$ defining type $T$, Autosubst generates the predicate $\text{InN}.T.F$.

For every instantiation $I$ with instantiated type $T$, Autosubst generates the modular induction principle $\text{induction}_T$ for $T$.

5.2 Modular Syntax with Binders

Autosubst 2 offers support for customised syntax with unscoped de Bruijn binders [9] and instantiation of parallel substitutions $\langle \_ \rangle : (N \rightarrow \text{exp}) \rightarrow \text{exp} \rightarrow \text{exp}$ based on the $\sigma$-calculus [1]. Substitutions are restricted to a finite set of primitives [1]: The identity substitution $\text{var}$, shifting $\uparrow$ which increases all variables by 1, extension $s, \sigma$ which extends a substitution $\sigma$ with $s$ at the first position, and last composition of substitutions.

Given a signature in HOAS input, Autosubst generates instantiation of renamings and substitutions to terms and a range of substitution lemmas. Later, the user can use $s(\xi)$ to denote the instantiation of a renaming, i.e. a substitution which only substitutes variables, and $s[\sigma]$ to denote the instantiation of a full substitution. Renamings range over $\xi$ and $\zeta$, while substitutions range over $\sigma$ and $\tau$. For example, $\beta$-reduction in the $\lambda$-calculus can be presented as:

\[
\text{app} \langle \lambda.s \rangle t > s[t, \text{var}].
\]

The user can moreover normalize expressions that contain substitutions via the $\text{asimpl}$ command. The procedure rewrites with the previously defined substitution lemmas. For the untyped $\lambda$-calculus this is a convergent [7], sound and complete decision procedure for equality [25], i.e. each valid assumption-free equation $s = t$ can be proven. See [28] for a more detailed description of binders and primitives in Autosubst 2.

Here, we extend these substitution primitives to modular syntax. As before, the user is untouched of all these internal changes. He can simply use the corresponding notation and automation tactics.

First, recall that there are two new syntactic categories: features (e.g. $\text{exp}_f$ or $\text{exp}_g$) and instantiated types (e.g. $\text{exp}_f$). We have to adapt all primitives of Autosubst (instantiation of renamings and substitutions on terms, substitution lemmas, automation) to these additions. There are two main changes: First, Autosubst 2 internally uses a dependency graph which has to be adapted. Second, we need to modularise functions and lemmas.

In general, the dependency analysis of Autosubst has to be extended to accommodate for the additional dependencies. We have several restrictions: Nothing may depend (i.e. include) a combined sort. Combined sorts depend on features, and a combined sort consists of features only. A variable feature is added automatically if any sort requires a negative occurrence. Independence of variable allows us to later import variables into different features.

Moreover, we adapted all functions and laws according to the changes promoted in the first part of the paper. For example, renaming on the $\lambda$-calculus has the following type:

\[
\langle \_ \rangle \text{exp} : (N \rightarrow N) \rightarrow \text{exp} \rightarrow \text{exp}
\]

We use smart constructors, assumptions in the feature functions and lemmas, and use smart constructors instead of the actual constructors. Also, the composition functions and laws have to be adapted, and we have to take more care of the exact dependencies. The $\text{asimpl}$ tactic now further includes the injectivity equations.

5.3 Dynamic Code Generation for Modular Syntax

A key part of the MetaCoq framework [26] is a monad which can be used to manipulate the environment in Coq. Monadic programs $P$ can be executed using the MetaCoq Run $P$ vernacular and can generate definitions, assume variables in sections, or open proof goals for the user.

To ease the definition and composition of modular functions and lemmas, we define several monadic programs and notations for them. We chose the notations such that the programs look like regular Coq commands defining functions and lemmas.

Support for the Definition of Functions and Lemmas.

We define the commands

\begin{verbatim}
MetaCoq Run Modular Fixpoint name where A extends B at n : type := body.
\end{verbatim}

and

\begin{verbatim}
MetaCoq Run Modular Lemma name where A extends B at n with [...; $f_i \rightsimeq g_i$;...] : type.
\end{verbatim}

$A$ and $B$ are types, $n$ is a natural number signalling on which argument the recursion will be on and $f_i \rightsimeq g_i$ signals that in the inductive hypothesis $f_i$ should be replaced by $g_i$. The implementation of both commands only differs in that...
the first command already takes the body of the function, whereas the latter opens a proof goal.

For a feature functor $F$ over a type $E$, the command for modular lemmas will generate the correct statement over the type $FE$ with an inductive hypothesis talking about $E$ and open a proof goal for the user. If modular induction lemmas are available, the inductive hypothesis mentions the $\epsilon$ predicate. Afterwards, it assumes the statement for $E$ as a variable, to make it available for subsequent lemmas defined in the section.

**Composition Support.** Lemmas can be composed using the command

```
MetaCoq Run Compose Lemma on n : T.
```

The command proves $1 : T$ by recursion over the $n$-th argument, followed by an application of the feature lemma $1_F$. What remains is to fill in dependencies feature lemmas, which can be done automatically by registering every lemma in a hint database immediately after proving it and then using the eauto tactic for all dependencies.

The `by induction using` modifier can be used to signal the application of a modular induction lemma $H$.

We define the command for functions as an alias.

5.4 Custom Tactics

We describe tactics simplifying the use of modular syntax.

- **msimpl.** This tactic simplifies goals using the injections for functions and predicates. These equations are registered in a hint database after their definition, together with retract equations.

- **minversion.** This tactic is an extension of Coq’s inversion tactic to modular syntax. It applies registered inversion lemmas, then uses Coq’s inversion tactic and resolves contradictory cases with the injectivity of inj.

- **mconstructor.** The tactic is a combination of msimpl and the constructor tactic.

5.5 Interactive Development of Modular Proofs

Given a HOAS input file, Autosubst outputs a range of files $F \cdot v$ for every feature $F$ (in the example from section 2, files corresponding to $\expvar$, $\expp$, $\expb$, and $\expn$) together with one file each for instantiated expressions ($\expv$, $\expp$, $\expb$, $\expn$).

If a user wants to add another feature or another instantiation, they change the HOAS input file and reruns the static code generation of Autosubst. The parts of the code for the existing features will stay unchanged, and a file for the new feature created, retaining true modularity.

Statements on features should then be proven in separate files, each importing the relevant feature file. Statements on instantiations can be obtained using our dynamic code generation after importing the statements for features and the file containing the instantiation.

```
the x-th element of \Gamma is A

\[
\Gamma \vdash s : A \rightarrow B \quad \Gamma \vdash t : B
\]
\[
\Gamma \vdash \lambda x : A. s : A \rightarrow B
\]
\[
\Gamma \vdash \lambda \lambda x : A. s : A \rightarrow B
\]
```

Figure 10. Typing and reduction for $\lambda$-expressions.

6 Case Study: Type Preservation, Weak Head Normalisation, and Strong Normalisation

We give an overview of our most involved case study. We extend the proof of preservation from Section 4 to cover $\expvar$ and $\expp$. By this introduction of binders, the preservation proof becomes a lot more involved.

We further show both weak head and strong normalisation for this calculus, which can be seen as a variant of mini-ML without fixpoints. We define expressions as combination of $\expvar$, $\expp$, $\expb$, and $\expn$. Besides $\tyb$ and $\tyn$ defined before we define a third modular type feature for mini-ML

```
Inductive ty_\lambda (ty : Type) := arr (\lambda : ty) (B : ty) : ty \lambda ty.
```

where we write $\text{arr } AB$ as $A \rightarrow B$. We use Autosubst 2 to automatically generate modular substitution functions and modular correctness lemmas for these types.

We assume that the reader is familiar with the standard proofs of preservation and normalisation. For weak head normalisation, we follow the technique by Dreyer et al. [11]; for strong normalisation, we follow the modular technique due to Schäfer [25], first used in [14]. These references also serve as introductions to the techniques used.

For the sake of brevity, we state modular lemmas only once. They are annotated by subscripts $v$, $\lambda$, $\beta$, and $\eta$, which can be clicked to see the Coq code corresponding to the proof for this feature. Similarly, the Lemma keyword can be clicked to access the code for the composed statement (all solved by our tactic). As before, modular lemmas have access to an induction hypothesis, which we do not make explicit. To denote the modular part of predicates, we use the subscript $i$, e.g. we write $\Gamma \vdash i : A$ for the modular definition of $\vdash$ in feature $i$.

We extend the typing predicate $\vdash$ and small-step reduction $\succ$ defined in Figures 7 and 8 to $\expvar$ and $\expp$ in Figure 10. We write $\text{nil}$ for the empty context, and $A, \Gamma$ for the context extended with a new type $A$. 
As in Section 4 we assume the following inclusions:

\[
\begin{align*}
\Gamma \vdash s : A & \rightarrow \Gamma \vdash s : A \quad (4) \\
\sigma \triangleright s & \rightarrow s \triangleright t \quad (5) \\
\Gamma \vdash \text{inj } s : A & \rightarrow \Gamma \vdash \text{inj } s : A \quad (6) \\
\text{inj } s \triangleright t & \rightarrow \text{inj } s \triangleright t \quad (7)
\end{align*}
\]

### 6.1 Type Preservation

We start with renaming and context morphism lemmas [15] for the typing predicate. We write \(\Gamma \lesssim \xi \Delta\), if \(\Delta\) is a reordering of \(\Gamma\) via the renaming \(\xi\), i.e. if \(\Delta(\xi \cdot x) = \Gamma x\) for all variables \(x < |\Gamma|\). This is a special case of a context morphism \(\Gamma <_{\sigma} \Delta\) on substitutions, where \(\Gamma \vdash x : A\) implies that \(\Delta \vdash \sigma x : A\).

Note that to state modular substitution lemmas (which are automatically generated by Autosubst) we need feature interaction, i.e. mention other features. Here, the statements for all features need to mention variables. Thus, we need to assume that exp\(_{\text{var}}\) \(\vdash\) exp \(\vdash\): for all features. For the context substitution lemma, we also need to know how typing behaves on variables, i.e. assume that \(\Gamma \vdash s : A \rightarrow \Gamma \vdash s : A\).

#### Lemma 6.1\(_{\text{ABN}}\)

If \(\Gamma \vdash s : A\) and \(\Gamma \lesssim \xi \Delta\), then \(\Delta \vdash s(\xi) : A\).

**Proof.** By induction on \(\Gamma \vdash s : A\). In the Coq proofs, we use the mconstructor tactic in each case to use the specific typing rules for all features. For abstraction, we use asimpl to simplify renamings.

#### Lemma 6.2\(_{\text{ABN}}\)

If \(\Gamma \vdash s : A\) and \(\Gamma <_{\sigma} \Delta\), then \(\Delta \vdash s(\sigma) : A\).

**Proof.** Analogous to Lemma 6.1. The proof for e.g. the \(\lambda\) case requires the typing rules for variables.

Both lemmas need the definition of renaming and substitution together with a wealth of structure. The asimpl tactic can solve all goals regarding renamings and substitution immediately. We come to preservation.

#### Lemma 6.3\(_{\text{ABN}}\)

If \(\Gamma \vdash s : A\) and \(s \triangleright t\), then \(\Gamma \vdash t : A\).

**Proof.** By induction on \(\Gamma \vdash s : A\), and a subsequent case analysis on \(s \triangleright t\) via minversion. In the case of abstraction, we require the context morphism Lemma 6.2.

### 6.2 Weak Head Normalisation

Every well-typed expression reduces to a weak-head normal form. We follow the proof outline by Dreyer et al. [11] and define a logical relation split into a value relation and an expression relation as well as the notion of semantic typing.

We define weak head normal forms as a modular function

\[
\text{whnf} : \text{exp} \rightarrow \text{Prop}.
\]

As before, we simultaneously assume the existence of a function \(\text{whnf}\) : \(\exp \rightarrow \text{Prop}\). Similar to predicates and types, we will need that the modularly defined parts behave like the overall function, i.e.

\[
\text{whnf} \ (\text{inj } s) = \text{whnf}_i (s) \quad (8)
\]

The value relation \(\mathcal{V}(A) : ty \rightarrow \exp \rightarrow \mathcal{P}\) is defined as a modular function by recursion on the type. We use set-like notation for better readability:

\[
\begin{align*}
\mathcal{V}_A (A \rightarrow B) & := \{ \lambda A.s \mid \forall i.v.v \in \mathcal{V}(A) \rightarrow \exists u.v'[\xi]. v' > u \wedge v' \in \mathcal{V}(B) \} \\
\mathcal{V}_B (\mathbb{B}) & := \{ \text{constBool}_{\xi} \text{true}, \text{constBool}_{\xi} \text{false} \} \\
\mathcal{V}_N (\mathbb{N}) & := \{ \text{constNat}_{\xi} n \mid n \in \mathbb{N} \}
\end{align*}
\]

Since the variable feature does not specify types and the function is by recursion on types, we do not define \(\mathcal{V}_{\text{var}} (A)\). To define \(\mathcal{V}_A (\_\_\_\_\_)\), we need a case analysis on the expression, which is only possible by first using the retraction function \(\text{retr}\) due to the type of the relation.

The expression relation, its lifting to contexts, and semantic typing do not require modularity. Instead, they can be defined completely parametrically in the relations \(\mathcal{V}\) and \(\triangleright\) in a global file.

\[
\begin{align*}
\mathcal{E} (A) & := \{ s \mid \exists s.s >^* u \wedge u \in \mathcal{V}(A) \} \\
\mathcal{G} (\Gamma) & := \{ \sigma \mid \forall x. \Gamma x = \text{Some } A \rightarrow \sigma x \in \mathcal{V}(A) \} \\
\Gamma \vdash s : A & := \forall \sigma. \sigma \in \mathcal{G} (\Gamma) \rightarrow s(\sigma) \in \mathcal{E} (A).
\end{align*}
\]

**Fact 6.4**

If \(s \in \mathcal{V}(A)\), then \(s(\xi) \in \mathcal{E}(A)\).

The following closure properties of reduction are proven per feature, but are not modular in the original sense, because they talk about the specific constructors.

**Lemma 6.5**

Let \(s >^* s', t >^* t',\) and \(u >^* u'\). Then

1. \(\text{app}_{\leq} s t >^* \text{app}_{\leq} s't'\)
2. \(\text{if}_{\leq} s \text{ then } t \text{ else } u >^* \text{ if}_{\leq} s' \text{ then } t' \text{ else } u'\)
3. \(s +_{\leq} t >^* s' +_{\leq} t'\).

**Proof.** By induction on \(s >^* s', t >^* t',\) and \(u >^* u'\). In each case, we use the mconstructor tactic.

We can now show monotonicity of the relation and that every element of a logical relation is in whnf:

**Lemma 6.6\(_{\text{ABN}}\)**

If \(s \in \mathcal{V}_i (A)\), then \(s(\xi) \in \mathcal{V}(\text{inj } A)\).

**Lemma 6.7\(_{\text{ABN}}\)**

If \(s \in \mathcal{V}_i (A)\), then \(\text{whnf } s\).

This suffices to modularly prove the fundamental lemma:

**Lemma 6.8\(_{\text{ABN}}\)**

If \(\Gamma \vdash s : A\), then \(\Gamma \vdash s : A\).

**Proof.** By induction on \(\Gamma \vdash s : A\). The proof requires Lemma 6.5 and repeatedly that the retract is tight. In the case of abstraction, we need Lemmas 6.7 and 6.6. For abstraction, we encounter the term

\[
s_{\{\text{var } 0, \sigma \circ \langle 1 \rangle \}|v, \xi \circ \text{var }}
\]

which we simplify to \(s_{\{v, \xi\}}\) using asimpl.
Weak head normalisation is then a non-modular consequence from the fundamental lemma:

**Lemma 6.9** If nil ⊢ s : A, then s ⊢ v for a v with whnf v.

*Proof.* If nil ⊢ s : A, we know that also nil ⊢ s : A. As var ∈ G (nil), also s[var] ∈ E (A), and s ∈ E (A) by the substitution laws, thus the claim holds. □

6.3 Strong Normalisation

Using Schäfer’s generalisation of the technique for weak head normalisation [25], the proof for strong normalisation is entirely analogous. We use an inductive definition of strongly normalizing terms

\[
\forall t. s > t \rightarrow sn(t) \rightarrow sn(s)
\]

and define the modular value and the (again) non-modular expression relation as follows:

\[
\mathcal{V}_1 (A \rightarrow B) := \{ \lambda. s | \forall \xi. v. v \in E (A) \rightarrow v'[\xi, \xi] \in E (B) \}
\]

\[
\mathcal{V}_B (\mathbb{B}) := \{ \text{constBool}_\mathbb{C}. \text{true}, \text{constBool}_\mathbb{C}. \text{false} \}
\]

\[
\mathcal{V}_\mathbb{N} (\mathbb{N}) := \{ \text{constNat}_\mathbb{N}. n \mid n : \mathbb{N} \}
\]

\[
\text{whnf } s \rightarrow s \in \mathcal{V}(A) \quad \forall t. s > t \rightarrow t \in E (A)
\]

We can prove the following property in general for E (\_):

**Fact 6.10** var x ∈ E (A).

Similar to before, we show monotonicity:

**Lemma 6.11,\_\_\_** If s ∈ \mathcal{V}_1 (A), then s(\xi) ∈ \mathcal{V}(A).

*Proof.* Analogous to Lemma 6.6. □

We will also need that steps are closed under substitution:

**Lemma 6.12,\_\_\_** If s >_\_\_ s', then s[\sigma] >_\_\_ s'[\sigma].

*Proof.* By induction on s >_\_\_ s', using the constructor tactic and the respective substitution properties. □

The last missing parts are two inversion-like properties of renamings. It is easy to show that renamings inversely preserve weak head normal forms:

**Lemma 6.13,\_\_\_** If whnf (s(\xi)), then whnf s.

We also would like to show that if a renamed term makes a step, the result can be written as a renamed term again. However, this property is not modular, as it depends on a global property of renamings which we have to assume and later prove globally:

**Lemma 6.14,\_\_\_** Assume that if s(\xi) = inj t then there is s' s.t. s = inj s. Then if s'(\xi) >_\_\_ t there is t' s.t. t = t'(\xi) ∧ s' >_\_\_ t.'

Finally, we need to prove properties of the expression relation:

**Lemma 6.15** The following hold:

1. If s ∈ E (A), then sn(s).
2. If s ∈ E (A) and s >_\_\_ t, then t ∈ E (A).
3. If s is a value and s ∈ E (A), then s ∈ \mathcal{V}(A).
4. If the relation is monotone, and values and reduction are stable under the anti-renaming, the closure is monotone as well.

*Proof.* (1)-(3) are immediate. (4) uses Lemma 6.13. □

We define the context relation and semantic typing analogously as in the weak head normalisation case. Again, we obtain a fully modular proof of the fundamental lemma and can conclude strong normalisation:

**Lemma 6.16,\_\_\_** If Γ ⊢ s : A, then Γ ⊢ s : A.

**Lemma 6.17** If Γ ⊢ s : A, then sn(s).

6.4 Evaluation

All files regarding this case study are in the SN directory.

Based on a HOAS specification of syntax for both expressions and types, Autosubst generates 800 lines of code, consisting of the definitions of feature constructors, smart constructors, substitutions, automation for substitution and the combined type exp (in file expressions.v).

The proofs of preservation, weak head normalisation and strong normalisation are done per feature, in files sn_ar.i.th.v (250 lines), sn_bool.v (255 lines), sn_lam.v (400 lines), and sn_var.v (90 lines), totalling to about 1000 lines of code. The code consists of about 45% of specification and 55% proofs.

The composed results, as well as the global lemmas for the combined type exp, are in sn.v, totalling to 190 lines, with about one-third specification. Note that this file also contains the non-modular proofs, e.g. on the expression relation.

7 Related Work

We compare our approach with the recent literature with a special focus on approaches that adapt Data Types à la Carte to proof assistants. All these approaches fulfil the criterion of true modularity.

**Data Types à la Carte.** Swierstra [29] introduces a practical approach to modular syntax in Haskell. As explained in Section 2, Data Types à la Carte is based on a general instantiatable expression type \( \text{Exp} : (\text{Type} \rightarrow \text{Type}) \rightarrow \text{Type} \), whereas we use fixed, static instantiations.

For function definitions, we do not rely on algebras, but directly use Coq’s built-in functions. We think that this improves both the transparency and accessibility of our code.

In Haskell, definitions are restricted to polymorphic, non-dependent types, and hence neither dependent functions nor dependent predicates are handled. However, the ideas scale, as demonstrated by our case studies.

Data Types à la Carte works with injections. Every injection in Haskell morally corresponds to a tight retract in Coq, which we require for our proofs.
Modular Type Safety Proofs in Agda. Schwaab and Siek [24] adapt the Data Types à la Carte approach to Agda and syntactically define a class of strictly positive functors which includes the identity functor, constant functors, products, and coproducts. A function `eval : Functor → Type → Type` is used to evaluate a functor and enables the definition of the least fixed point over strictly positive functors. Due to a more restricted checker for strict positivity, the approach is inapplicable to Coq (see Figure 12).

We were unable to obtain the source code for the paper, which makes a comparison difficult. However, we were able to implement the case study, proving preservation for a language with natural numbers, arrays and options (but no means for case analysis and no abstractions or binders) in about 150 lines of code. Schwaab and Siek mention that their final proof which composes all features is rejected in Agda because termination cannot be verified. In our setting, this does not pose a problem and Coq checks termination instantaneously.

Meta-Theory à la Carte. Delaware, d S Oliveira, and Schrijvers [10] adapt the Data Types à la Carte approach to Coq via Mendler-style Church encodings. Church-encodings rely on Coq’s impredicative sets option and are used as a replacement of inductive data types.

Their framework is implemented entirely in Coq and consists of 2500 lines. As a case study, they prove monotonicity and type soundness of a big-step presentation of mini-ML. The definition of mini-ML using our HOAS input language is depicted in Figure 11. They define evaluation as a function taking a natural-number step index (often called fuel) and circumvent the need for substitutions with environments. They also support binders, using a PHOAS approach [6], but do not use it in their case study. One key challenge in the case study is feature interaction, surfacing as the need to assume inversion properties.

For each feature, typing, evaluation, monotonicity, and type preservation require about 1100 lines of code. We implemented the same case study for comparison. With our approach, all five features together need about 625 lines of code, i.e. we need about 125 lines per feature while obtaining transparent statements. For a more detailed discussion of this big line difference, see the next paragraph.

The indirectness induced by Church encodings replacing inductive types, algebras replacing functions and proof algebras replacing proofs impairs the readability of definitions for non-experts impacting transparency and accessibility.

Furthermore, Coq’s impredicative `Set` option is known to be inconsistent with classical logic (excluded middle plus unique choice) and makes constructors of some inductive types lose injectivity.

Generic Datatypes à la Carte. Keuchel and Schrijvers [17] present a solution with binders based on a universe of containers. Containers consist of a type of codes and an interpretation function mapping codes to types. Their framework needs about 3500 lines of code. Keuchel and Schrijvers use the same case study as Delaware et al., i.e. monotonicity and preservation of mini-ML. Their framework is similar in size and needs 1050 lines per feature of the case study, resulting in 5150 lines of code in total.

From a theoretical perspective, the usage of containers seems to be the most satisfying approach, since it subsumes, for example, the strictly positive functors used by Schwaab and Siek [24]. From a practical perspective, using codes is unsatisfying, since definitions become even harder to read.

Recall further that with our approach we used a mere fraction of the lines of codes (125 lines per feature/625 loc in total) of both Delaware et al. and Keuchel and Schrijvers. For the composition of lemmas, there is no difference in lines.

We see three main reasons: First, our approach requires less preliminary code (which seems to be around 1/5th of the code needed in Generic Data Types à la Carte). Secondly, the main difficulty in the case study is the inclusion of fixpoints. Our approach handles a more efficient treatment of this inclusion, discussed in detail below, which seems to spare us around 1/5th of their lines again. The remaining difference is due to the directness of our approach, resulting in fewer lines for function definitions, proofs and tactics. In Generic Data Types à la Carte, types and predicates have to be encoded as polynomial functors or containers, which is not needed in our approach at all.

We now discuss the simplification regarding the inclusion of fixpoints in our approach. The monotonicity of a step-indexed evaluation function

\[ \text{eval} : \mathbb{N} \rightarrow \text{env} \rightarrow \text{exp} \rightarrow \text{option value} \]

can be stated as

\[ \text{eval } n \ E \ e = \text{Some } v \rightarrow m \geq n \rightarrow \text{eval } m \ E \ e = \text{Some } v. \]

For the inclusion of fixpoints, we have two choices: We choose to introduce a new value, namely recursive closures. This enforces us to change the evaluation rule for applications for the new fixpoint feature, which is conceptually no problem and still allows to reuse proofs.

Unfortunately, this is no option for both Meta-Theory à la Carte and Generic Data Types à la Carte. Since rules can seemingly not be changed afterwards, they have to re-utilise non-recursive closures, which are usually the values of non-recursive abstractions. Evaluating a fixpoint with step index `n` then results in unfoldng the fixpoint `n` times and using the resulting closure as value. For step index `m \geq n`, the unfolding will now not be the same, because it is unfolded more often. They thus have to prove a changed statement talking about `n`-approximations of values and environments, making all proofs considerably harder and longer.

---

5See file `TypeSafetyAgda/TypeSafety.v`.
6See directory `GDTC`.
7https://github.com/coq/coq/wiki/Impredicative-Set
Proof Reuse. A variety of approaches has investigated proof reuse in general. An exhaustive historical overview is available in Section 6.4 of Ringer et al.’s survey on proof engineering [21]. Approaches in the literature span from implementing dedicated proof assistants [12], via extensions of type theory [4] to automated approaches to generalising statements as much as possible [20].

Regarding modular syntax, Mulhern [18] uses heuristical automation for Coq written in OCaml to combine proofs for small languages over closed inductive expression types to more extensive languages by combining the types of the small languages automatically to one big type.

Boite [5] implements OCaml commands for Coq to extend types and predicates by parameters and constructors. Proofs over extended predicates then reuse proofs of the original form with a tactic that transforms proof terms and requires Coq to re-check the proof term. In principle, the commands could be implemented in MetaCoq.

Johnsen and Lüth [16] implement proof term transformations for LCF-style proof terms in Isabelle. Ringer et al. [22] implement a restricted form of proof reuse in Coq, but not geared towards modular syntax. Recent research into exploiting instances of univalence to obtain equivalences for free in Coq [30] may strengthen this approach and potentially adapt it to modular syntax.

8 Discussion and Future Work

In this paper, we have suggested a practical approach to modular syntax: a user specifies syntax modularly using a HOAS input language, and we generate feature functors and instantiations combining features in the spirit of the Data Types à la Carte approach, together with automation for the treatment of binders. We further provide commands to define modular recursive functions, state modular lemmas, and combine modular constructions fully automatically.

We implemented a variety of case studies:

• Type preservation for a language with natural numbers, arrays and options, which is the case study used by Schwaab and Siek [24]. This proof takes about 150 lines of code. We were unable to obtain the code of Schwaab and Siek for comparison.

• Monotonicity and type preservation for a big-step presentation of mini-ML (i.e. simply-typed λ-calculus with natural numbers, arithmetic, booleans and recursive abstractions), which is the case study used by Delaware et al. [10] and Keuchel and Schrijvers [17]. We need 625 lines for the implementation, compared to 5500 and 5250 lines, respectively.

• Type preservation, weak head normalisation and strong normalisation for a small-step presentation for the simply-typed λ-calculus with natural numbers and booleans, which is similar to one of the case studies posed as part of the POPLMark Reloaded challenge [2] in about 1200 lines.

Based on the case studies, we evaluate our approach concerning the evaluation criteria from the introduction:

• Conciseness: Using our approach only has a moderate overhead over writing non-modular code. Our modular tactics make proofs similar to non-modular proofs, also with regard to their length. A user does not have to write any preliminary code to use our approach. There is however some overhead when defining modular dependent predicates because we do not support MetaCoq commands for this yet.

• Transparency: Compared to related work, our approach benefits from its directness. Composed types directly correspond to their non-modular counterparts, and we can hence omit manual adequacy proofs. We can use Coq’s standard commands to define functions, fixpoints and lemmas and do not have to rely on algebras or proof algebras. Hence our code reads basically like non-modular code. After composition, one does not need to be familiar with our approach at all to understand definitions and statements.

• Accessibility: Due to its simplicity, we believe that the learning curve for our approach is relatively flat. Any Coq user who has mechanised meta-theory proofs before should be able to adapt to our approach quickly. While the approach is usable without any tool support, Autosubst’s and MetaCoq’s automation eases the formalisation of syntax with binders, also in our modular setting.

• True Modularity: Both with and without modular induction lemmas, Coq does not have to re-check proof terms. Without modular induction lemmas, termination has to be re-checked if a proof is instantiated several times, which is still considerably faster than type-checking.

In future work, we want to extend our automation. First, we plan to extend Autosubst’s code generation for modular types to scoped syntax, since currently we only support unscoped de-Bruijn syntax in the modular setting. Second, we would like to extend the input language to be also able to define dependent predicates, which so far requires manual proofs. Last, we would like to implement MetaCoq commands to define and compose dependent predicates.

We would also like to evaluate our approach in more case studies. The POPLMark Reloaded challenge poses a proof of strong normalisation as in our case study, but with a different proof strategy. It would be interesting to see whether this can be proven fully modularly. Once Autosubst supports patterns, we would like to try to give a modular solution for the full POPLMark challenge. We want to try to modularise big, existing developments: for example, extending the mechanised results for call-by-push-value in [14] to the results from [13].

Finally, the ultimate test whether our approach can be considered practical will be whether external, third-party proof developments will use it. We think that the theorem proving community would greatly benefit from more modular developments and look forward to their input.
References


[19] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tat-


[28] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tat-


A Figures

Figure 11. Definition of mini-ML.
Inductive Functor := Id : Functor | Const : Type → Functor.
Fixpoint eval (X : Type) (F : Functor) : Type :=
  match F with| Id ⇒ X | Const Y ⇒ X end.
Inductive mu (F : Functor) := inn : eval (mu F) F → mu F.
(* Error: Non strictly positive occurrence of "mu" *)
(* in "eval (mu F) F → mu F". *)

Figure 12. Failing definition in Coq.