

# 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. 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.

**CCS Concepts** • Theory of computation → Lambda calculus; Type theory; • Software and its engineering → Syntax.

**Keywords** modular syntax, syntax with binders, Coq

### ACM Reference Format:

Yannick Forster and Kathrin Stark. 2020. Coq à la Carte: A Practical Approach to Modular Syntax with Binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP ’20)*, January 20–21, 2020, New Orleans, LA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org). CPP ’20, January 20–21, 2020, New Orleans, LA, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7097-4/20/01...\$15.00

<https://doi.org/10.1145/3372885.3373817>

ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3372885.3373817>

### 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 [6, 11, 20, 23, 29] 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* [40]:

“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 [36] 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) +  $\mathbb{B}$  -- if and boolean constants
Lambda X =  $\mathbb{N}$  + (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  $\text{Exp}$  via an arbitrary functor is impossible in proof assistants such as Coq or Agda, which require defined types to be strictly positive [9].

To the best of our knowledge, all adaptations of Data Types à la Carte to proof assistants hence add a layer of indirectness to circumvent this problem: Schwaab and Siek [29] formalise the syntax of (some) strictly positive functors and only allow such instances,<sup>1</sup> other approaches work with Church encodings [11] or containers and proof algebras [20].

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.<sup>2</sup> Finally, a user has to learn about the encoding, for example, how to define 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 variants  $\text{Exp}_{F_1}, \dots, \text{Exp}_{F_n}$  suffice to define modular functions, inductive modular predicates over modular syntax, and modular proofs over these definitions. This approach already yields concise, transparent, accessible, and truly modular developments in Coq.

In the second step, we increase the usability of this approach and automatically generate these variants. We extend the Autosubst 2 tool [35] to allow modular specifications of syntax. The user chooses which set of features  $F_i$  should be present in a variant, 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 [31] to program both input and composition mechanisms for modular functions and lemmas.

The main conceptual limitation we are aware of is that our approach does not support changing the types of modularly defined types and predicates a posteriori, i.e. it is not possible to parameterise a modularly defined syntax type over a set of indices while introducing a new feature. Other limitations, like the omission of mutual recursion, are not conceptual and discussed in the last section for future work.

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 [29]. Our development seems to be similar in size.
- Compatibility with instantiation with renamings 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, but called mini-ML consistently in related work on modular syntax), which is the case study used in [11, 20]. 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 the Data Types à la Carte approach [36] 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 [35] 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 [31]. 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.

<sup>1</sup>The exact definition is impossible in Coq, see Section 7.

<sup>2</sup>Alternatively, the connection to the theorem can be proven by hand, a property often called adequacy, which, however, does not work well with the goal of conciseness.

**Development.** Our development is available online at <https://github.com/uds-psl/coq-a-la-carte-cpp20>.

Section 6.4, which was added for the final version of this paper, reuses parts of the second author’s thesis [34].

## 2 Modular Syntax

In this section, we give a high-level overview of modular syntax via our adaption of the Data Types à la Carte approach [36] to Coq<sup>3</sup>. For this introduction, we spell out and 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 [10]:

$$s, t : \text{exp}_1 ::= \text{var } x \mid \text{app } s t \mid \lambda. s \quad x \in \mathbb{N}$$

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:

$$s, t, u : \text{exp}_2 ::= \dots \mid b \mid \text{if } s \text{ then } t \text{ else } u$$

$$s, t : \text{exp}_3 ::= \dots \mid n \mid s + t$$

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 [36] defines extensible expressions as

```
Inductive Exp (F : Type → Type) :=
| In : F (Exp F) → Exp F.
```

together with feature functors, for our example defined in Figure 1.<sup>4</sup> As a convention, specific feature functors always have a symbol as a subscript, e.g.  $\text{exp}_\lambda$ .

With these definitions and the pointwise coproduct on functors, written  $:+$ , we can define different variants of  $\text{exp}$  as:

```
Definition exp1 := Exp (expλ :+: expvar).
Definition exp2 := Exp (expλ :+: expvar :+: expℬ).
Definition exp3 := Exp (expλ :+: expvar :+: expℕ).
```

<sup>3</sup>The corresponding Coq code is in the files `Counting/section2_count.v` and `Counting/section2_count_metacoq.v`.

<sup>4</sup>We treat variables as a separate feature to be more flexible concerning different features using variables.

```
Inductive expvar (exp : Type) :=
| var : ℕ → expvar exp.

Inductive expλ (exp : Type) :=
| app : exp → exp → expλ exp
| abs : exp → expλ exp.

Inductive expℬ (exp : Type) :=
| constBool : ℬ → expℬ exp
| if : exp → exp → exp → expℬ exp.

Inductive expℕ (exp : Type) : Type :=
| plus : exp → exp → expℕ exp
| constNat : ℕ → expℕ exp.
```

Figure 1. Feature functors.

```
Inductive exp1 :=
| injvar : expvar exp1 → exp1
| injλ : expλ exp1 → exp1.

Inductive exp2 :=
| injvar : expvar exp2 → exp2
| injλ : expλ exp2 → exp2
| injℬ : expℬ exp2 → exp2.

Inductive exp3 :=
| injvar : expvar exp3 → exp3
| injλ : expλ exp3 → exp3
| injℕ : expℕ exp3 → exp3.
```

Figure 2. Generated variants.

However, Coq’s positivity checker rejects the definition of  $\text{Exp}$ , because it would introduce a logical inconsistency via the negation functor  $F(T) := T \rightarrow \perp$ .<sup>5</sup>

Instead of a generalised fixed-point type  $\text{Exp} : (\text{Type} \rightarrow \text{Type}) \rightarrow \text{Type}$ , our approach simply inlines the above definitions, see Figure 2. We write  $\text{exp}_1$ ,  $\text{exp}_2$ , and  $\text{exp}_3$  for different variants 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 [24] specification language where syntax and features can be defined. To generate the above types, one would use the HOAS signature depicted in Figure 3. Negative occurrences of types (here the first  $\text{exp}$  in the type of  $\text{abs}$ ) are translated to binders in the output. Each feature is surrounded by a `begin... end` block, each variant can be generated via the `compose` command.

### 2.2 Recursive Functions on Modular Syntax

Our modular definition of expressions allows us to define modular functions. As a simple example, we define a modular

<sup>5</sup>  $\text{Exp}F \leftrightarrow (\text{Exp}F \rightarrow \perp)$  would be provable, but is contradictory.

```

exp,  $\mathbb{B}$ ,  $\mathbb{N}$ : Type

begin lam
  abs : (exp  $\rightarrow$  exp)  $\rightarrow$  exp
  app : exp  $\rightarrow$  exp  $\rightarrow$  exp
end lam

begin arith
  constNat :  $\mathbb{N}$   $\rightarrow$  exp
  plus :  $\mathbb{N}$   $\rightarrow$   $\mathbb{N}$   $\rightarrow$  exp
end arith

begin booleans
  constBool :  $\mathbb{B}$   $\rightarrow$  exp
  if : exp  $\rightarrow$  exp  $\rightarrow$ 
exp  $\rightarrow$  exp
end booleans

compose lambdas := lam
compose booleans :=
  lam :+: bool
compose arith :=
  lam :+: arith

```

**Figure 3.** Example input file for exp.

```

Section lam.
Variable exp: Type.
Variable  $\lfloor \_ \rfloor$  : exp  $\rightarrow$   $\mathbb{N}$ .
Definition  $\lfloor \_ \rfloor_\lambda$  : exp $_\lambda$  exp  $\rightarrow$   $\mathbb{N}$  :=
  fun e  $\Rightarrow$  match e with
  |  $\lambda$ .s  $\Rightarrow$  |s|
  | app s t  $\Rightarrow$  |s| + |t|
  end.
End lam.

```

**Figure 4.** Definition of the counting function for the lambda feature.

function  $\lfloor \_ \rfloor$  : exp  $\rightarrow$   $\mathbb{N}$  which counts the number of leaves (i.e. variables or constants) in an expression. We first focus on exp<sub>2</sub> and modularly add the definitions for exp<sub>3</sub> later on.

The definition consists of two steps: First, the definition of the feature functions, then their composition.

We parameterise the feature functions by a type exp and a function  $\lfloor \_ \rfloor$  : exp  $\rightarrow$   $\mathbb{N}$  and then define them as

$$\begin{aligned} \lfloor \_ \rfloor_{\text{var}} &: \text{exp}_{\text{var}} \text{exp} \rightarrow \mathbb{N} \\ \lfloor \_ \rfloor_\lambda &: \text{exp}_\lambda \text{exp} \rightarrow \mathbb{N} \\ \lfloor \_ \rfloor_{\mathbb{B}} &: \text{exp}_{\mathbb{B}} \text{exp} \rightarrow \mathbb{N} \end{aligned}$$

for all features contained in exp<sub>2</sub>. Figure 4 shows the exemplary definition of  $\lfloor \_ \rfloor_\lambda$ . In Coq, we add the parameters using the `Variable` command in a section. After closing the section, we obtain for instance a function  $\lfloor \_ \rfloor_\lambda$  of type  $\forall \text{exp}, (\text{exp} \rightarrow \mathbb{N}) \rightarrow \text{exp}_\lambda \text{exp} \rightarrow \mathbb{N}$ , i.e. a function which is parameterised in the type exp and the function  $\lfloor \_ \rfloor$ . We write applications to a function  $f$  and an expression  $e$  as  $|e|_f^f$ . The technique to add the function used for the recursive call as a parameter is also referred to as “open recursion”.

In the second step and as shown in Figure 5, the counting function for exp<sub>2</sub> can be obtained by a simple case analysis calling the respective feature functions. Since e.g.  $\lfloor \_ \rfloor_\lambda$  only uses  $\lfloor \_ \rfloor$  on structurally smaller arguments, this definition is terminating and is accepted by Coq’s termination checker.

```

Fixpoint  $\lfloor \_ \rfloor$  (e : exp2) :  $\mathbb{N}$  :=
  match e with
  | injvar e  $\Rightarrow$  |e|var $\lfloor \_ \rfloor$ 
  | inj $_\lambda$  e  $\Rightarrow$  |e| $_\lambda$  $\lfloor \_ \rfloor$ 
  | inj $_{\mathbb{B}}$  e  $\Rightarrow$  |e| $_{\mathbb{B}}$  $\lfloor \_ \rfloor$ 
  end.

```

**Figure 5.** Definition of counting function.

**Tool support.** We provide syntax for both the definition and combination of modular fixpoints. Instead of the definition of  $\lfloor \_ \rfloor_\lambda$  in Figure 4, a user can write:

```

MetaCoq Run
Modular Fixpoint  $\lfloor \_ \rfloor_\lambda$  where exp $_\lambda$  exp extends exp with  $\lfloor \_ \rfloor$  :=
  fun (s : exp $_\lambda$  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  $\lfloor \_ \rfloor$  on 0 : exp2  $\rightarrow$   $\mathbb{N}$ .

```

The 0 indicates that the fixpoint is defined by recursion on the first argument. The commands are implemented in MetaCoq (which explains the MetaCoq Run prefix), we elaborate on this in Section 5.

### 2.3 Proofs on Modular Syntax

We further develop proofs over modular syntax; as they are just dependently typed functions, this is analogous to the last section. As an example, we show that every expression has leaves, i.e.  $|s| > 0$ .

To do so, we add the following parameter to the section:

```

Variable count_gt :  $\forall e : \text{exp}, |e| > 0$ .

```

and then show the statement for the separate features:

```

Lemma count_gtvar :
   $\forall e : \text{exp}_{\text{var}} \text{exp}, |e|_{\text{var}} > 0$ . Proof. (* ... *) Defined.
Lemma count_gt $_\lambda$  :
   $\forall e : \text{exp}_\lambda \text{exp}, |e|_\lambda > 0$ . Proof. (* ... *) Defined.
Lemma count_gt $_{\mathbb{B}}$  :
   $\forall e : \text{exp}_{\mathbb{B}} \text{exp}, |e|_{\mathbb{B}} > 0$ . Proof. (* ... *) Defined.

```

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

$$|\text{app } s \ t|_{\mathbb{B}} = |s| + |t| > 0$$

where  $|s|$  and  $|t|$  are larger than 0 by the assumption count\_gt, and so the whole claim follows.

The lemma for e.g. the variant exp<sub>2</sub> now follows immediately from the respective lemmas for exp $_\lambda$  and exp $_{\mathbb{B}}$ :

```

Fixpoint count_gt (e : exp2) : |s| > 0.
Proof.
  destruct e; cbn;
  [ apply count_gtvar | apply count_gtλ | apply count_gtB ];
  eauto.
Qed.

```

Since Coq’s induction principle for  $\text{exp}_2$  is too weak, we do the proof by direct recursion<sup>6</sup> on the expression  $e$  rather than induction. We fix this deficiency in Section 3 and introduce modular induction principles.

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

```

MetaCoq Run
Modular Lemma count_gtλ where expλ exp extends exp at 0
with [ |λ ~> | ] : ∀ s, |s| > 0.

```

Stating this is equivalent to the following two commands:

```

Variable count_gt : ∀ s, |s| > 0.
Lemma count_gtλ : ∀ s, |s| > 0.

```

To combine lemmas, a user can write:

```

MetaCoq Run Compose
Lemma count_gt on 0 : ∀ (s : exp2), |s| > 0.

```

## 2.4 Modular Constructors

We want to lift the constructors from features as  $\text{app}$  to constructors for a variant, e.g.  $\text{exp}_2$ . *Modular constructors* (called smart constructors in [36]) combine the constructors of the modular type  $\text{exp}_B$  with the actual constructors of  $\text{exp}_2$ , i.e.

```

Definition appB s t := injλ (app s t).

```

However, more variants like  $\text{exp}_3$  again lead to code duplication. We mirror *Swierstra* with tight retracts between types, defined in Coq using type classes [33] as follows:

```

Class X <: Y :=
{ inj : X → Y; retr : Y → option X;
  retract_works : ∀ x, retr (inj x) = Some x;
  retract_tight : ∀ x y, retr y = Some x →
  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:

```

Instance exp_retractλ : expλ exp <: exp.
Instance exp_retractB : expB exp <: exp.

```

<sup>6</sup>In the Coq proofs, this requires that  $\text{count\_gt}_\lambda$  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} (expλ exp <: exp) s t := inj (app s t).

```

Similarly, we define constructors  $\text{if}_< \_ \text{then } \_ \text{else } \_$  and  $\text{var}_<$  and  $\text{constBool}_<$  to use in arbitrary contexts:

```

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

```

**Tool support.** `Autosubst` automatically generates the proofs for  $\text{exp\_retract}_\lambda$ ,  $\text{exp\_retract}_B$ , and the definition of modular constructors, e.g.  $\text{app}_<$ ,  $\text{var}_<$ , and  $\text{if}_< \_ \text{then } \_ \text{else } \_$ .

## 2.5 Introduction of a New Feature

If we extend our definitions to the type  $\text{exp}_3$ , we have to define  $|\_|\mathbb{N}$  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:

```

Section Arith.
Variable exp : Type.

```

```

MetaCoq Run Modular
Fixpoint |_|\mathbb{N} where (expN exp) extends exp with |_|\mathbb{N} :=
fun (s : expN exp) =>
  match s with
  | constNat _ => 1
  | plus s t => 1 + |s| + |t|
  end.

```

```

MetaCoq Run Modular
Lemma count_gtN where (expN exp) extends exp
with [ |_|\mathbb{N} ~> |_|\mathbb{N} ] : ∀ s, |s|\mathbb{N} > 0.
Next Obligation. (* ... *) Defined.
End Arith.

```

```

MetaCoq Run Compose Fixpoint |_|\mathbb{N} on 0 : ∀ (s : exp3), \mathbb{N}.
MetaCoq Run Compose Lemma count_gt on 0 : ∀
(s : exp3), |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 [36]. 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 variants 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  $\text{exp}_2$  reads as follows:

```

exp2_ind :
  ∀ P : exp2 → Prop,

```

$$\begin{aligned}
 e \in_{\text{var}} \text{var } x &:= \perp \\
 e \in_{\lambda} \text{app } s \ t &:= e = s \vee e = t \\
 e \in_{\lambda} \lambda. s &:= e = s \\
 e \in_{\mathbb{B}} \text{constBool } b &:= \perp \\
 e \in_{\mathbb{B}} (\text{if } s \text{ then } t \text{ else } u) &:= e = s \vee e = t \vee e = u
 \end{aligned}$$

**Figure 6.** Containment for expressions.

$$\begin{aligned}
 (\forall e : \text{exp}_{\text{var}} \text{exp}_2, P (\text{inj}_{\text{var}} e)) &\rightarrow \\
 (\forall e : \text{exp}_{\lambda} \text{exp}_2, P (\text{inj}_{\lambda} e)) &\rightarrow \\
 (\forall e : \text{exp}_{\mathbb{B}} \text{exp}_2, P (\text{inj}_{\mathbb{B}} e)) &\rightarrow \forall e : \text{exp}_2, P e
 \end{aligned}$$

Note that there are no inductive hypotheses available. We fix this problem by automatically generating a stronger induction principle as part of the output of Autosubst. Using the induction principle avoids re-checking of termination in instantiated proofs like `count_gt`.

The strong induction principles are built on a notion of syntactic *subexpressions*, defined in Figure 6. For example,  $s$  is a syntactic subexpression of `app s t`, written  $s \in_{\lambda} \text{app } s \ t$ .

**Theorem 3.1** `exp2_induction` :

$$\begin{aligned}
 \forall P : \text{exp}_2 \rightarrow \text{Prop}, \\
 (\forall e : \text{exp}_{\text{var}} \text{exp}_2, (\forall e' \in_{\text{var}} e. P e') \rightarrow P (\text{inj}_{\text{var}} e)) &\rightarrow \\
 (\forall e : \text{exp}_{\lambda} \text{exp}_2, (\forall e' \in_{\lambda} e. P e') \rightarrow P (\text{inj}_{\lambda} e)) &\rightarrow \\
 (\forall e : \text{exp}_{\mathbb{B}} \text{exp}_2, (\forall e' \in_{\mathbb{B}} e. P e') \rightarrow P (\text{inj}_{\mathbb{B}} e)) &\rightarrow \\
 \forall e : \text{exp}_2, P e
 \end{aligned}$$

*Proof.* By recursion on  $e$ .  $\square$

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  $(\forall e' \in_{\text{var}} e. |e'| > 0)$ , then  $|e|_{\text{var}} > 0$ .
2. If  $(\forall e' \in_{\lambda} e. |e'| > 0)$ , then  $|e|_{\lambda} > 0$ .
3. If  $(\forall e' \in_{\mathbb{B}} e. |e'| > 0)$ , then  $|e|_{\mathbb{B}} > 0$ .

**Lemma 3.3** For all  $e : \text{exp}_2$ ,  $|e| > 0$ .

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

## 4 Modular Predicates

We extend our approach to modular inductive predicates with dependent types over modular syntax. We first define a type system and a nondeterministic reduction relation for terms in  $\text{exp}_{\mathbb{N}}$  and  $\text{exp}_{\mathbb{B}}$  and extend it to  $\text{exp}_{\text{var}}$  and  $\text{exp}_{\lambda}$  in Section 6.

As types, we use natural numbers and booleans:

$$\begin{array}{c}
 \frac{}{\Gamma \vdash_{\mathbb{N}} \text{atom} < n : \mathbb{N}} \qquad \frac{\Gamma \vdash s : \mathbb{N} \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash_{\mathbb{N}} s + < t : \mathbb{N}} \\
 \frac{}{\Gamma \vdash_{\mathbb{B}} \text{constBool} < b : \mathbb{B}} \qquad \frac{\Gamma \vdash b : \mathbb{B} \quad \Gamma \vdash e_1 : A \quad \vdash e_2 : A}{\Gamma \vdash_{\mathbb{B}} \text{if} < b \text{ then } e_1 \text{ else } e_2 : A}
 \end{array}$$

**Figure 7.** Typing for arithmetic and boolean expressions.

$$\begin{array}{c}
 \frac{s > s'}{s + < t >_{\mathbb{N}} s' + < t} \qquad \frac{t > t'}{s + < t >_{\mathbb{N}} s + < t'} \\
 \frac{}{\text{atom} < m + < \text{atom} < n >_{\mathbb{N}} \text{atom} < (m + n)} \\
 \frac{}{\text{if} < \text{constBool} < \text{true} \text{ then } e_1 \text{ else } e_2 >_{\mathbb{B}} e_1} \\
 \frac{}{\text{if} < \text{constBool} < \text{false} \text{ then } e_1 \text{ else } e_2 >_{\mathbb{B}} e_2} \\
 \frac{e_1 > e'_1}{\text{if} < e_1 \text{ then } e_2 \text{ else } e_3 >_{\mathbb{B}} \text{if} < e'_1 \text{ then } e_2 \text{ else } e_3} \\
 \frac{e_2 > e'_2}{\text{if} < e_1 \text{ then } e_2 \text{ else } e_3 >_{\mathbb{B}} \text{if} < e_1 \text{ then } e'_2 \text{ else } e_3} \\
 \frac{e_3 > e'_3}{\text{if} < e_1 \text{ then } e_2 \text{ else } e_3 >_{\mathbb{B}} \text{if} < e_1 \text{ then } e_2 \text{ else } e'_3}
 \end{array}$$

**Figure 8.** Reduction for arithmetic and boolean expressions.

$$\frac{\Gamma \vdash_{\mathbb{N}} s : A}{\Gamma \vdash s : A} \qquad \frac{\Gamma \vdash_{\mathbb{B}} s : A}{\Gamma \vdash s : A} \qquad \frac{s >_{\mathbb{N}} s'}{s > s'} \qquad \frac{s >_{\mathbb{B}} s'}{s > s'}$$

**Figure 9.** Typing and reduction for full expressions.

Inductive `tyN` :=  $\mathbb{N}$ .  
 Inductive `tyB` :=  $\mathbb{B}$ .

We define modular typing relations  $\_ \vdash_{\mathbb{N}} \_ : \_$  and  $\_ \vdash_{\mathbb{B}} \_ : \_$  of type `list ty  $\rightarrow$  exp  $\rightarrow$  ty  $\rightarrow$  Prop` (where contexts are modeled as lists) for both features as well as modular evaluation relations  $\_ >_{\mathbb{N}} \_$  and  $\_ >_{\mathbb{B}} \_ : \text{exp} \rightarrow \text{exp} \rightarrow \text{Prop}$ . As before, the relations are parameterised by a type `exp` and the following two relations:

$$\begin{aligned}
 \_ \vdash \_ : \_ : \text{list ty} \rightarrow \text{exp} \rightarrow \text{ty} \rightarrow \text{Prop} \\
 \_ > \_ : \text{exp} \rightarrow \text{exp} \rightarrow \text{Prop}
 \end{aligned}$$

We also have to parameterise by retracts as before, e.g. assume  $\text{exp}_{\mathbb{N}} \text{exp} < : \text{exp}$ . As a side-effect of this, we use smart constructors everywhere. From now on, we switch to a more mathematical presentation. The corresponding Coq code can be accessed by clicking on the statements. The definitions of the relations are in Figures 7 and 8.

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 injections, because for predicates the proof itself is irrelevant (as long as there is a proof):

$$\Gamma \vdash_i s : A \rightarrow \Gamma \vdash s : A \tag{1}$$

$$s >_i t \rightarrow s > t \tag{2}$$

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

$$\Gamma \vdash \text{inj } s : A \rightarrow \Gamma \vdash_i \text{inj } s : A \quad (3)$$

We now give a modular proof of type preservation for this language. We want to show that if  $\Gamma \vdash s : A$  and  $s > t$ , then  $\Gamma \vdash t : A$  by induction on  $s > t$ . Similar to before, we use the modular versions  $>_i$  in the modular statements for arguments we want to do induction on. Otherwise, we always use the full versions. The remainder of the proof is then analogous to the proofs in the last section:

✦ **Lemma 4.1** *Assume that if  $\Gamma \vdash s : A$  and  $s > t$ , then  $\Gamma \vdash t : A$ .*

1. *If  $\Gamma \vdash s : A$  and  $s >_{\mathbb{N}} t$ , then  $\Gamma \vdash t : A$ .*
2. *If  $\Gamma \vdash s : A$  and  $s >_{\mathbb{B}} t$ , then  $\Gamma \vdash t : A$ .*

*Proof.* We show the claim for arithmetic expressions by case analysis on  $s >_{\mathbb{N}} t$ . As  $\Gamma \vdash_{\mathbb{N}} s : A$  via eq. (3), we can do an inversion on the derivation. For  $\Gamma \vdash t : A$  it suffices to show that  $\Gamma \vdash_{\mathbb{N}} t : A$  via eq. (4), and so the claim holds.  $\square$

In the mechanised Coq proof, we use the tactics `minversion` and `mconstructor` to use Equations (3) to (5). It is again easy to deduce preservation for the variants, defined as follows:

```
Inductive exp := injN : expN exp → exp | injB : expB exp →
exp.
Inductive ty := injtyN : tyN ty → ty | injtyB : tyB ty → ty.
```

The relations for variants are in Figure 9.

✦ **Theorem 4.2** *If  $\Gamma \vdash s : A$  and  $s > t$ , then  $\Gamma \vdash t : A$ .*

*Proof.* By induction on  $s > t$  and Lemma 4.1.  $\square$

Preservation for the lambda feature requires the handling of binders, and we refer its discussion to Section 6.1.

## 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 [35] to support modular types. Based on this input, we implement static code-generation of feature functors and variants 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 `asimpl` tactic simplifying substitution goals also on modular syntax.

And third, we implement dynamic code generation based on MetaCoq [31] 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 [35] interface to modular types. Recall Figure 3 for an example input. Autosubst generates functors, types, retractions, modular constructors, and induction principles based on this input.

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

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

**Retracts.** For each specified variant  $I$  and every type  $T_1, \dots, T_m$  defined in a feature  $F$  of  $I$ , Autosubst proves that  $T\_F T <: T$ .

**Smart Constructors.** For each constructor  $C$ , Autosubst automatically defines the respective smart constructor called  $C\_$  via injections.

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

For every variant  $I$  with instantiated type  $T$ , Autosubst generates the definition of syntactic subexpressions and the modular induction principle `inductionT` for  $T$ .

### 5.2 Modular Syntax with Binders

Autosubst 2 offers support for customised syntax with pure de Bruijn binders [10] and instantiation with parallel substitutions  $[_] : (\mathbb{N} \rightarrow \text{exp}) \rightarrow \text{exp} \rightarrow \text{exp}$  based on the  $\sigma$ -calculus [1]. Substitutions are restricted to a finite set of primitives [1]: The identity substitution `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 with renamings and substitutions to terms and a range of substitution lemmas. Later, the user can use  $s\langle\xi\rangle$  to denote the instantiation with a renaming, i.e. a substitution which only substitutes variables, and  $s[\sigma]$  to denote the instantiation with 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:

$$\text{app } (\lambda.s) t > s[t, \text{var}].$$

The user can moreover normalise expressions that contain substitutions via the `asimpl` command. The procedure rewrites with the previously defined substitution lemmas. For the untyped  $\lambda$ -calculus, this is a convergent [8], sound and complete decision procedure for equality [28], i.e. each valid assumption-free equation  $s = t$  can be proven. See [35]

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. They can simply use the corresponding notation and automation tactics.

First, recall that there are two new syntactic categories: features (e.g.  $\text{exp}_\lambda$  or  $\text{exp}_\mathbb{B}$ ) and variants (e.g.  $\text{exp}_2$ ). We have to adapt all primitives of Autosubst (instantiation with 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, instantiation with renamings in the  $\lambda$ -calculus has the following type:

$$\langle \_ \rangle_{\text{exp}} : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \text{exp}_\lambda \text{ exp} \rightarrow \text{exp}$$

We use smart constructors, parameters 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 `asimpl` tactic now further includes the injectivity equations.

### 5.3 Dynamic Code Generation for Modular Syntax

A key part of the MetaCoq framework [31] 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 such as 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

MetaCoq Run Modular

`Fixpoint` name where A extends B at  $n$  : type := body.

and

MetaCoq Run Modular

`Lemma` name where A extends B at  $n$  **with** [...:  $f_i \rightsquigarrow g_i$ ;...]:type.

Here, A and B are types,  $n$  is a natural number signalling on which argument the recursion will be on and  $f_i \rightsquigarrow g_i$  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, similar to the respective non-modular commands in Coq. We do not generate assumptions that the modular function agrees with the non-modular version automatically and leave it for the user to add where needed.

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  $\in$  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 1` on  $n$  : T.

The command proves  $1 : T$  by recursion over the  $n$ -th argument, followed by an application of the feature lemma `1F`. 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 H` 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. The implementation of the tactics is relatively simple, but they make using modular syntax more convenient and make proof scripts look closer to their non-modular counterparts.

**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 extends 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  $F.v$  for every feature  $F$  (in the example from section 2, files corresponding to  $\text{exp}_{\text{var}}$ ,  $\text{exp}_\lambda$ ,  $\text{exp}_\mathbb{B}$ , and  $\text{exp}_\mathbb{N}$ ) together with one file each for instantiated expressions ( $\text{exp}_1$ ,  $\text{exp}_2$ ,  $\text{exp}_3$ ).

If a user wants to add another feature or another variant, 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, entailing true modularity.

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

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

We give an overview of our most involved case study. We first extend the proof of preservation from Section 4 to cover  $\text{exp}_{\text{var}}$  and  $\text{exp}_{\lambda}$ . By the 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  $\text{exp}_{\text{var}}$ ,  $\text{exp}_{\lambda}$ ,  $\text{exp}_{\mathbb{B}}$ , and  $\text{exp}_{\mathbb{N}}$ . Besides  $\text{ty}_{\mathbb{B}}$  and  $\text{ty}_{\mathbb{N}}$  defined before we define a third modular type feature for mini-ML

Inductive  $\text{ty}_{\lambda}$  ( $\text{ty} : \text{Type}$ ) := arr ( $A : \text{ty}$ ) ( $B : \text{ty}$ ) :  $\text{ty}_{\lambda}$   $\text{ty}$ .

where we write  $\text{arr } A B$  as  $A \rightarrow B$ . We use Autosubst 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 usual technique via a Kripke-style logical relation [17, 21, 38] and split the logical relation into an expression relation and a value relation following Dreyer et al. [13]; for strong normalisation, we follow the modular technique due to Schäfer [30], first used in [16]. 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  $v$ ,  $\lambda$ ,  $\mathbb{B}$ , and  $\mathbb{N}$ , which can be clicked on 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  $\Gamma \vdash_i s : A$  for the modular definition of  $\vdash$  in feature  $i$ .

We extend the typing predicate  $\vdash$  and small-step reduction  $>$  defined in Figures 7 and 8 to  $\text{exp}_{\text{var}}$  and  $\text{exp}_{\lambda}$  in Figure 10. We write  $\text{nil}$  for the empty context, and  $A, \Gamma$  for the context extended with a new type  $A$ .

$$\begin{array}{c} \text{the } x\text{-th element of } \Gamma \text{ is } A \\ \hline \Gamma \vdash_{\text{var}} \text{var } x : A \\ \hline \Gamma \vdash s : A \rightarrow B \quad \Gamma \vdash t : B \\ \hline \Gamma \vdash_{\lambda} s t : B \qquad \frac{A, \Gamma \vdash s : B}{\Gamma \vdash_{\lambda} \lambda A.s : A \rightarrow B} \\ \hline \frac{}{\text{app}_{<} (\lambda_{<} A.s) t >_{\lambda} s [t..]_{<}} \qquad \frac{s > s'}{\text{app}_{<} s t >_{\lambda} \text{app}_{<} s' t} \\ \hline \frac{t > t'}{\text{app}_{<} s t >_{\lambda} \text{app}_{<} s t'} \qquad \frac{s > s'}{\lambda_{<} A.s >_{\lambda} \lambda_{<} A.s'} \end{array}$$

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

As in Section 4 we assume the following implications:

$$\Gamma \vdash_i s : A \rightarrow \Gamma \vdash s : A \quad (4)$$

$$s >_i t \rightarrow s > t \quad (5)$$

$$\Gamma \vdash \text{inj } s : A \rightarrow \Gamma \vdash_i \text{inj } s : A \quad (6)$$

$$\text{inj } s > t \rightarrow \text{inj } s >_i t \quad (7)$$

### 6.1 Type Preservation

We start with renaming and context morphism lemmas [18] for the typing predicate. We write  $\Gamma \vdash \xi : \Delta$ , if  $\Delta$  is a reordering of  $\Gamma$  via the renaming  $\xi$ , i.e. if  $\Delta(\xi x) = \Gamma x$  for all variables  $x < |\Gamma|$ . This is a special case of a context morphism  $\Gamma \vdash \sigma : \Delta$  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. Hence, all features are parameterised over variables. Thus, we need to assume that  $\text{exp}_{\text{var}} \text{exp } < : \text{exp}$  for all features. For the context substitution lemma, we also need to know how typing behaves on variables, i.e. assume that  $\Gamma \vdash_{\text{var}} s : A \rightarrow \Gamma \vdash s : A$ .

✦ **Lemma 6.1<sub>vλBN</sub>** *If  $\Gamma \vdash_i s : A$  and  $\Gamma \vdash \xi : \Delta$ , then  $\Delta \vdash s \langle \xi \rangle : A$ .*

*Proof.* By induction on  $\Gamma \vdash_i s : A$ . 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.  $\square$

✦ **Lemma 6.2<sub>vλBN</sub>** *If  $\Gamma \vdash_i s : A$  and  $\Gamma \vdash \sigma : \Delta$ , 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.  $\square$

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<sub>vλBN</sub>** *If  $\Gamma \vdash_i s : A$  and  $s > t$ , then  $\Gamma \vdash t : A$ .*

*Proof.* By induction on  $\Gamma \vdash_i s : A$ , and a subsequent case analysis on  $s > t$  via minversion. In the case of abstraction, we require the context morphism Lemma 6.2.  $\square$

## 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. [13] 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

$$\begin{aligned} \text{whnf}_{\text{exp\_var}}(\text{var } x) &:= \perp \\ \text{whnf}_\lambda(\lambda \_ \_.) &:= \top & \text{whnf}_{\lambda \_} &:= \perp \\ \text{whnf}_{\mathbb{B}}(\text{constBool } b) &:= \top & \text{whnf}_{\mathbb{B} \_} &:= \perp \end{aligned}$$

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

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

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

$$\begin{aligned} \mathcal{V}_\lambda(A \rightarrow B) &:= \{\lambda A.s \mid \forall \xi v.v \in \mathcal{V}(A) \rightarrow \\ &\quad \exists v'.s[v, \xi] >^* v' \wedge v' \in \mathcal{V}(B)\} \\ \mathcal{V}_{\mathbb{B}}(\mathbb{B}) &:= \{\text{constBool}_{<} \text{true}, \text{constBool}_{<} \text{false}\} \\ \mathcal{V}_{\mathbb{N}}(\mathbb{N}) &:= \{\text{constNat}_{<} n \mid n : \mathbb{N}\} \end{aligned}$$

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  $\mathcal{V}_\lambda(\_)$ , 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 (which is lifted in the above definition of the value 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  $>$  in a global file.

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

**Fact 6.4** *If  $s \in \mathcal{V}(A)$ , then  $s \in \mathcal{E}(A)$ .*

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.  $\text{app}_{<} s t >_{<}^* \text{app}_{<} s' t'$
2.  $\text{if}_{<} s \text{ then } t \text{ else } u >_{<}^* \text{if}_{<} s' \text{ then } t' \text{ else } u'$
3.  $s +_{<} t >_{<}^* s' +_{<} t'$

*Proof.* By induction on  $s >^* s'$ ,  $t >^* t'$ , and  $u >^* u'$ . In each case, we use the  $\text{mconstructor}$  tactic.  $\square$

We can now show that the logical relation is compatible with instantiation with renamings  $\xi$  and that every element of a logical relation is in  $\text{whnf}$ :

**Lemma 6.6**<sub>vλBN</sub> *If  $s \in \mathcal{V}_i(A)$ , then  $s\langle\xi\rangle \in \mathcal{V}(\text{inj } A)$ .*

**Lemma 6.7**<sub>vλBN</sub> *If  $s \in \mathcal{V}_i(A)$ , then  $\text{whnf } s$ .*

This suffices to modularly prove the fundamental lemma:

**Lemma 6.8**<sub>vλBN</sub> *If  $\Gamma \vdash_i s : A$ , then  $\Gamma \vDash s : A$ .*

*Proof.* By induction on  $\Gamma \vdash_i s : A$ . The proof uses 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

$$s[\text{var } 0, \sigma \circ \langle\uparrow\rangle][v, \xi \circ \text{var}]$$

which simplifies to  $s[v, \sigma \circ \langle\xi\rangle]$  using  $\text{asimpl}$ .  $\square$

Weak head normalisation is then a non-modular consequence from the fundamental lemma:

**Lemma 6.9** *If  $\text{nil} \vdash s : A$ , then  $s >^* v$  for a  $v$  with  $\text{whnf } v$ .*

*Proof.* If  $\text{nil} \vdash s : A$ , we know that also  $\text{nil} \vDash s : A$ . As  $\text{var} \in \mathcal{G}(\text{nil})$ , also  $s[\text{var}] \in \mathcal{E}(A)$ , and  $s \in \mathcal{E}(A)$  by the substitution laws, thus the claim holds.  $\square$

## 6.3 Strong Normalisation

Using Schäfer's generalisation of the technique for weak head normalisation [30], the proof for strong normalisation is entirely analogous. We use an inductive definition of strongly normalising terms

$$\frac{\forall t. s > t \rightarrow \text{sn}(t)}{\text{sn}(s)}$$

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

$$\begin{aligned} \mathcal{V}_\lambda(A \rightarrow B) &:= \{\lambda A.s \mid \forall \xi v.v \in \mathcal{E}(A) \rightarrow s[v, \xi] \in \mathcal{E}(B)\} \\ \mathcal{V}_{\mathbb{B}}(\mathbb{B}) &:= \{\text{constBool}_{<} \text{true}, \text{constBool}_{<} \text{false}\} \\ \mathcal{V}_{\mathbb{N}}(\mathbb{N}) &:= \{\text{constNat}_{<} n \mid n : \mathbb{N}\} \\ \text{whnf } s \rightarrow s \in \mathcal{V}(A) &\quad \forall t.s > t \rightarrow t \in \mathcal{E}(A) \\ &\quad \underline{\hspace{10em}} \\ &\quad s \in \mathcal{E}(A) \end{aligned}$$

We can prove the following property in general for  $\mathcal{E}(\_)$ :

**Fact 6.10**  $\text{var } x \in \mathcal{E}(A)$ .

Similar to before, we show compatibility with instantiation with renamings:

**Lemma 6.11**<sub>vλBN</sub> *If  $s \in \mathcal{V}_i(A)$ , then  $s\langle\xi\rangle \in \mathcal{V}(A)$ .*

*Proof.* Analogous to Lemma 6.6.  $\square$

We will also need that steps are closed under substitution:

**Lemma 6.12**<sub>vλBN</sub> *If  $s >_i s'$ , then  $s[\sigma] > s'[\sigma]$ .*

*Proof.* By induction on  $s >_i s'$ , using the  $\text{mconstructor}$  tactic and the respective substitution properties.  $\square$

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**<sub>vλBN</sub> *If  $\text{whnf}(s\langle\xi\rangle)$ , then  $\text{whnf } s$ .*

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**<sub>vλBN</sub> *Assume that if  $s\langle\xi\rangle = \text{inj } t$  then there is  $s' \text{ s.t. } s = \text{inj } s'$ . Then if  $s'\langle\xi\rangle \succ_i t$  there is  $t' \text{ s.t. } t = t'\langle\xi\rangle \wedge s' \succ t'$ .*

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

**Lemma 6.15** *The following hold:*

1. *If  $s \in \mathcal{E}(A)$ , then  $\text{sn}(s)$ .*
2. *If  $s \in \mathcal{E}(A)$  and  $s \succ t$ , then  $t \in \mathcal{E}(A)$ .*
3. *If  $s$  is a value and  $s \in \mathcal{E}(A)$ , then  $s \in \mathcal{V}(A)$ .*
4. *If the relation is compatible with instantiation with renamings, and values and reduction are stable under the anti-renaming, the closure is compatible with instantiation with renamings 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**<sub>vλBN</sub> *If  $\Gamma \vdash_i s : A$ , then  $\Gamma \vDash s : A$ .*

✦ **Lemma 6.17** *If  $\Gamma \vdash s : A$ , then  $\text{sn}(s)$ .*

## 6.4 Modularity

Not all parts can be defined in a modular manner. We distinguish between *modular definitions* which are proven once for each feature, *parameterised definitions* which are stated before all definitions in a parameterised fashion, and last, *global definitions*, which require global knowledge. See Figure 11 for an overview on which definitions and lemmas are proven in which manner.

Most lemmas and specifically those that are by induction on a respective modular predicate can be proven modularly. Parameterised definitions could in principle be proven once for each feature, but this would contain repetitive code. These are definitions which are independent of the particular feature, e.g. the lifting of the logical relation to expressions or contexts. Last, there is one proof for which we were unable to find a modular proof: This proof is the anti-renaming lemma for reduction, which also causes problems during substitution automation. This proof requires full knowledge of reduction which goes beyond the properties of a tight retract.

What	Mod.	Param.	Global
Substitution boilerplate	x	-	-
Typing	x	-	-
Reduction	x	-	-
CRL	x	-	-
CML	x	-	-
Preservation	x	-	-
LR for WN	x	-	-
Monotonicity LR	x	-	-
Lifting of LR	-	x	-
Value inclusion	-	x	-
Congruence	x	-	-
Fundamental lemma	x	-	-
WN	-	x	-
LR for SN	x	-	-
Monotonicity LR	x	-	-
Closure properties	-	x	-
Substitutivity reduction	x	-	-
Anti-renaming reduction	-	-	x
Fundamental lemma SN	x	-	-
SN	-	x	-

**Figure 11.** Overview of modular proofs for preservation, weak head normalisation, and strong normalisation [34].

## 6.5 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 functors, smart constructors, substitutions, automation for substitution and the variant exp (in file expressions.v).

The proofs of preservation, weak head normalisation and strong normalisation are done per feature, in files sn\_arith.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 variant 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 [36] 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 variants.

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 [29] 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 only applicable to Coq by a relational definition of eval.

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.<sup>7</sup> 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 [11] 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 12. 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 [7], 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

```
ty, exp,  $\mathbb{B}$ ,  $\mathbb{N}$ : Type
begin NatCase
natCase : exp → exp →
exp → exp
end NatCase

begin Arith
natT : ty
constNat :  $\mathbb{N}$  → exp
plus :  $\mathbb{N}$  →  $\mathbb{N}$  → exp
end Arith

begin Booleans
boolT : ty
constBool :  $\mathbb{B}$  → exp
if : exp → exp → exp →
exp
end Booleans

begin NatCase
natCase : exp → exp →
exp → exp
end NatCase

begin Lambdas
arr : ty → ty → ty
lam : ty → exp → exp
app : exp → exp → exp
end

begin Recursion
fixp : ty → exp
end Recursion
```

Figure 12. Definition of mini-ML.

transparent statements<sup>8</sup> 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 of 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<sup>9</sup>) and makes constructors of some inductive types lose injectivity.

In our approach, we support, however, one dimension of modularity less, compared to MTC. MTC and also the follow-up work by Delaware et al. [12] allow the types of functions and lemmas to change when adding features. Currently, we do not know how to incorporate this into our approach and leave it to future work.

**Generic Datatypes à la Carte.** Keuchel and Schrijvers [20] 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 [29]. 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

<sup>7</sup>See file [TypeSafetyAgda/TypeSafety.v](#).

<sup>8</sup>See directory [GDTTC](#).

<sup>9</sup><https://github.com/coq/coq/wiki/Impredicative-Set>

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.

It is also possible that our generation of boilerplate code concerning binders and substitutions saves lines – the specific case study is, however, implemented using environments and does not use substitutions.

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 unfolding 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.

**Open Inductive Predicates in Isabelle.** Molitor [22] implements open inductive predicates in Isabelle. Open (i.e. modular) theorems can be proved, but modular syntax is not covered. Since the approach does not support modular theorems to depend on previous modular theorems, it is hard to reimplement the case studies used in this work in the Isabelle framework, making a direct comparison of feasibility impossible. However, the design choices for the user interface might help to guide the development of a mature tool in the future.

**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 [26]. Approaches in the literature span from implementing dedicated proof assistants [14], via extensions of type theory [4] to automated approaches to generalising statements as much as possible [25].

Regarding modular syntax, Mulhern [23] 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 [6] 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 [19] implement proof term transformations for LCF-style proof terms in Isabelle. Ringer et al. [27] 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 [37] may strengthen this approach and potentially adapt it to modular syntax.

#### **Generation of containment and induction principles.**

Our generation of modular induction principles is closely related to the generation of strong induction principles using ELPI by [39], where our containment relation is a special case of the unary parametricity translation used by Tassi. Instead of using Autosubst's code generation, we could probably have used Coq-Elpi – but did not want to use yet another external tool.

The Equations package by Sozeau and Mangin [32] can also generate induction principles in Coq, based on function definitions.

For more complex (e.g. mutual) types, the containment predicate becomes harder and we might have to follow a route similar to the one taken by Blanchette et al. [5].

## 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-like input language, and we generate feature functors and variants 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 [29]. 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  $\lambda$ -calculus with natural numbers, arithmetic, booleans and recursive abstractions), which is the case study used by Delaware et al. [11] and Keuchel and Schrijvers [20]. 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  $\lambda$ -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 both our approach and our automation. Regarding the approach, we would like to investigate whether it can be adapted to allow changing the types of e.g. modular functions when a new feature is introduced, as supported by Meta-Theory à la Carte.

Regarding the automation, we first plan to extend Autosubst's code generation for modular types to scoped syntax, since currently we only support pure 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 [16] to the results from [15].

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.

## Acknowledgments

We would like to thank Dominik Kirst, Fabian Kunze, Steven Schäfer, and Gert Smolka for their constructive feedback on the presentation of the material. We thank Simon Spies for a helpful initial discussion on different approaches for modular syntax and the anonymous reviewers for their insightful comments and suggestions.

## References

- [1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. 1991. Explicit substitutions. *Journal of functional programming* 1, 4 (1991), 375–416.
- [2] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark Reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019).
- [3] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *TPHOLS*, Vol. 3603. Springer, 50–65.
- [4] Gilles Barthe and Olivier Pons. 2001. Type isomorphisms and proof reuse in dependent type theory. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 57–71.
- [5] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. 2014. Truly modular (co) datatypes for Isabelle/HOL. In *International Conference on Interactive Theorem Proving*. Springer, 93–110.
- [6] Olivier Boite. 2004. Proof reuse with extended inductive types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 50–65.
- [7] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, Vol. 43. ACM, 143–156.
- [8] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. 1996. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM (JACM)* 43, 2 (1996), 362–397.
- [9] Haskell B Curry. 1942. The inconsistency of certain formal logics. *The Journal of Symbolic Logic* 7, 3 (1942), 115–117.
- [10] Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*

- (*Proceedings*) 75, 5 (1972), 381 – 392.
- [11] Benjamin Delaware, Bruno C d S Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 207–218.
- [12] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno CdS Oliveira. 2013. Modular monadic meta-theory. *ACM SIGPLAN Notices* 48, 9 (2013), 319–330.
- [13] Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. 2018. Semantics of Type Systems – Lecture Notes. (2018). <https://plv.mpi-sws.org/semantics/2017/lecturenotes.pdf>
- [14] Amy Felty and Douglas Howe. 1994. Generalization and reuse of tactic proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 1–15.
- [15] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 13.
- [16] Yannick Forster, Steven Schäfer, Simon Spies, and 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*. ACM, 118–131.
- [17] Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and types*. Vol. 7. Cambridge University Press Cambridge.
- [18] Healfdene Goguen and James McKinna. 1997. Candidates for substitution. *LFCS report series-Laboratory for Foundations of Computer Science ECS LFCS* (1997).
- [19] Einar Broch Johnsen and Christoph Lüth. 2004. Theorem reuse by proof term transformation. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 152–167.
- [20] Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*. ACM, 13–24.
- [21] John C Mitchell and Eugenio Moggi. 1991. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic* 51, 1-2 (1991), 99–124.
- [22] Richard Molitor. 2015. Open Inductive Predicates. Master’s thesis. Karlsruher Institut für Technologie (KIT).
- [23] Anne Mulhern. 2006. Proof weaving. In *Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*.
- [24] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN’88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. ACM, 199–208.
- [25] Olivier Pons. 2000. Generalization in type theory based proof assistants. In *International Workshop on Types for Proofs and Programs*. Springer, 217–232.
- [26] Talia Ringer, Karl Palmkog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281.
- [27] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for Proof Reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [28] Steven Schäfer, Gert Smolka, and Tobias Tebbi. 2015. Completeness and Decidability of de Bruijn Substitution Algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*. Springer-Verlag, Berlin, Heidelberg, 67–73. <https://doi.org/10.1145/2676724.2693163>
- [29] Christopher Schwaab and Jeremy G Siek. 2013. Modular type-safety proofs in Agda. In *Proceedings of the 7th workshop on Programming languages meets program verification*. ACM, 3–12.
- [30] Steven Schäfer. 2019. *Engineering Formal Systems in Constructive Type Theory*. Ph.D. Dissertation. Saarland University. <https://www.ps.uni-saarland.de/~schaefers/thesis/>
- [31] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2019. The MetaCoq Project. (2019).
- [32] Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 86.
- [33] Matthieu Sozeau and Nicolas Oury. 2008. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 278–293.
- [34] Kathrin Stark. 2019. *Mechanising Syntax with Binders in Coq*. Ph.D. Dissertation. Saarland University, Submitted.
- [35] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions. *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019* (2019).
- [36] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.
- [37] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for free: Univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 92.
- [38] William W Tait. 1967. Intensional interpretations of functionals of finite type I. *The journal of symbolic logic* 32, 2 (1967), 198–212.
- [39] Enrico Tassi. 2019. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O’Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 29:1–29:18. <https://doi.org/10.4230/LIPIcs.ITP.2019.29>
- [40] Philip Wadler et al. 1998. The expression problem. *Posted on the Java Genericity mailing list* (1998).