The Autosubst Library

Kathrin Stark
Heriot-Watt University, Edinburgh, UK

EuroProofNet Workshop WG 4
Sept 15 - 16 2025

Today’s Talk

1. Autosubst – a library for binders
2. Coq à la Carte - Towards a library for modular proofs

Today’s Talk

1. **Autosubst – a library for binders**
2. Coq à la Carte - Towards a library for modular proofs

How much do you need to know about the formalization of binders to write proofs about programming languages in a proof assistant?

Mechanized Metatheory for the Masses: The POPLMARK Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Abstract. How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?
We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus
with seco
marks em
lenging t
syntactic
and proof
benchmar
for compa

– *Binding.* Most programming languages have some form of binding in their syntax and require a treatment of α-equivalence and substitution in their semantics. To adequately represent many languages, the representation strategy must support multiple kinds of binders (e.g. term and type), constructs introducing multiple binders over the same scope, and potentially unbounded lists of binders (e.g. for record patterns).

Syntax/Variables in a Textbook

1.1.1. DEFINITION. Let

$$V = \{v_0, v_1, \dots\}$$

denote an infinite alphabet. The set Λ^- of *pre-terms* is the set of strings defined by the grammar:

$$\Lambda^- ::= V \mid (\Lambda^- \ \Lambda^-) \mid (\lambda V \ \Lambda^-)$$

1.1.11. DEFINITION. For $M \in \Lambda^-$ define the set $\mathrm{FV}(M) \subseteq V$ of *free variables* of M as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda x.P) & = & \mathrm{FV}(P) \backslash \{x\}; \\ \mathrm{FV}(P\ Q) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

If $\mathrm{FV}(M) = \{\}$ then M is called *closed.*

1.1.13. DEFINITION. For $M, N \in \Lambda^-$ and $x \in V$, the *substitution of N for x in M,* written $M[x := N] \in \Lambda^-$, is defined as follows, where $x \neq y$:

$$\begin{array}{lll} x[x := N] & = N; & \\ y[x := N] & = y; & \\ (P\ Q)[x := N] & = P[x := N]\ Q[x := N]; & \\ (\lambda x.P)[x := N] & = \lambda x.P; & \\ (\lambda y.P)[x := N] & = \lambda y.P[x := N], & \text{if } y \notin \mathrm{FV}(N) \text{ or } x \notin \mathrm{FV}(P); \\ (\lambda y.P)[x := N] & = \lambda z.P[y := z][x := N], & \text{if } y \in \mathrm{FV}(N) \text{ and } x \in \mathrm{FV}(P). \end{array}$$

Define preterms...

Sørenson, Urzyczyn - Lectures on the Curry-Howard Isomorphism

Syntax/Variables in a Textbook

1.1.15. DEFINITION. Let *α-equivalence,* written $=_\alpha$, be the smallest relation on Λ^-, such that

$$\begin{array}{ll} P =_\alpha P & \text{for all } P; \\ \lambda x.P =_\alpha \lambda y.P[x := y] & \text{if } y \notin \mathrm{FV}(P), \end{array}$$

and closed under the rules:

$$\begin{array}{lcll} P =_\alpha P' & \Rightarrow & \forall x \in V: & \lambda x.P =_\alpha \lambda x.P'; \\ P =_\alpha P' & \Rightarrow & \forall Z \in \Lambda^-: & P\ Z =_\alpha P'\ Z; \\ P =_\alpha P' & \Rightarrow & \forall Z \in \Lambda^-: & Z\ P =_\alpha Z\ P'; \\ P =_\alpha P' & \Rightarrow & & P' =_\alpha P; \\ P =_\alpha P' \ \&\ P' =_\alpha P'' & \Rightarrow & & P =_\alpha P''. \end{array}$$

... α-equivalence...

1.1.17. DEFINITION. Define for any $M \in \Lambda^-$, the *equivalence class* $[M]_\alpha$ by:

$$[M]_\alpha = \{N \in \Lambda^- \mid M =_\alpha N\}$$

Then define the set Λ of *λ-terms* by:

$$\Lambda \ = \ \Lambda^-/=_\alpha \ = \ \{[M]_\alpha \mid M \in \Lambda^-\}$$

... actual terms ...

1.1.18. WARNING. The notion of a pre-term and the associated explicit distinction between pre-terms and λ-terms introduced above are not standard in the literature. Rather, it is customary to call our pre-terms λ-terms, and then informally remark that α-equivalent λ-terms are "identified."

Sørenson, Urzyczyn - Lectures on the Curry-Howard Isomorphism

Syntax/Variables in a Textbook

1.1.19. NOTATION. We write M instead of $[M]_\alpha$ in the remainder. This leads to ambiguity: is M a pre-term or a λ-term? In the remainder of these notes, M should always be construed as $[M]_\alpha \in \Lambda$, *except when explicitly stated otherwise.*

1.1.20. DEFINITION. For $M \in \Lambda$ define the set $\mathrm{FV}(M) \subseteq V$ of *free variables* of M as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda x.P) & = & \mathrm{FV}(P)\backslash\{x\}; \\ \mathrm{FV}(P\ Q) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

If $\mathrm{FV}(M) = \{\}$ then M is called *closed.*

1.1.21. REMARK. According to Notation 1.1.19, what we really mean by this is that we define FV as the map from Λ to subsets of V satisfying the rules:

$$\begin{array}{lcl} \mathrm{FV}([x]_\alpha) & = & \{x\}; \\ \mathrm{FV}([\lambda x.P]_\alpha) & = & \mathrm{FV}([P]_\alpha)\backslash\{x\}; \\ \mathrm{FV}([P\ Q]_\alpha) & = & \mathrm{FV}([P]_\alpha) \cup \mathrm{FV}([Q]_\alpha). \end{array}$$

Strictly speaking we then have to demonstrate there there is at most one such function (uniqueness) and that there is at least one such function (existence).

Uniqueness can be established by showing for any two functions FV_1 and FV_2 satisfying the above equations, and any λ-term, that the results of FV_1 and FV_2 on the λ-term are the same. The proof proceeds by induction on the number of symbols in any member of the equivalence class.

To demonstrate existence, consider the map that, given an equivalence class, picks a member, and takes the free variables of that. Since any choice of member yields the same set of variables, this latter map is well-defined, and can easily be seen to satisfy the above rules.

In the rest of these notes such considerations will be left implicit.

... and definitions on terms.

Sørenson, Urzyczyn - Lectures on the Curry-Howard Isomorphism

… and that's without the proofs!

Goal: Prove substitutivity of reduction, i.e. that $s \succ t$ implies $s[\sigma] \succ t[\sigma]$.

$$
\begin{aligned}
s[\sigma][t[\sigma]..] &= s[\mathrm{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle][t[\sigma] \cdot \mathrm{var}] \\
&= s[(\mathrm{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathrm{var})]] && \text{compositionality} \\
&= s[(\mathrm{var}\,0)[t[\sigma] \cdot \mathrm{var}] \cdot (\sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathrm{var})]] && \text{distributivity} \\
&= s[(\mathrm{var}\,0)[t[\sigma] \cdot \mathrm{var}] \cdot \sigma(\langle\uparrow\rangle \circ [t[\sigma] \cdot \mathrm{var}])] && \text{associativity} \\
&= s[(t[\sigma] \cdot \mathrm{var})\,0 \cdot (\sigma \circ [\uparrow (t[\sigma] \cdot \mathrm{var})])] && \text{compositionality} \\
&= s[t[\sigma] \cdot (\sigma \circ [\mathrm{var}])] && \cdot\text{, interaction} \\
&= s[t[\sigma] \cdot \sigma] && \text{right identity} \\
&= s[t[\sigma] \cdot (\mathrm{var} \circ [\sigma])] && \text{left identity} \\
&= s[(t \cdot \mathrm{var}) \circ [\sigma]] && \text{distributivity} \\
&= s[t \cdot \mathrm{var}][\sigma]. && \text{compositionality}
\end{aligned}
$$

Which lemmas do we need?

How to apply them?

How much do you need to know about the formalization of binders to write proofs about programming languages in a proof assistant?

- How to state lemmas in this presentation?
- Which lemmas on substitution should I prove?
- How do I prove them?
- How do I use them?

POPLMark's Criteria for Success

The primary metric of success (beyond correctness, of course) is that a solution should give us confidence of future success of other formalizations carried out using similar techniques. In particular, this implies that:

- *The technology should impose reasonable overheads.* We accept that there is a cost to formalization, and our goal is *not* to be able to prove things more easily than by hand (although that would certainly be welcome). We are willing to spend more time and effort to use the proof infrastructure, but the overhead of doing so must not be prohibitive. (For example, as we discuss below, our experience is that explicit de Bruijn-indexed representations of variable binding structure fail this test.)
- *The technology should be transparent.* The representation strategy and proof assistant syntax should not depart too radically from the usual conventions familiar to the technical audience, and the content of the theorems themselves should be apparent to someone not deeply familiar with the theorem proving technology used or the representation strategy chosen.
- *The technology should have a reasonable cost of entry.* The infrastructure should be usable (after, say, one semester of training) by someone who is knowledgeable about programming language theory but not an expert in theorem prover technology.

Autosubst

A **choice** of
- Binder representation
- Definition of substitutions
- Set of substitution lemmas

that requires minimum input from the user

+

A **library format** to automatically generate the corresponding boilerplate, including automation tactic
that requires minimum input from the user

Schäfer, Tebbi, Smolka – CPP ‘15
S, Schäfer, Kaiser – CPP ‘19

There's a Variety of Binding Techniques in a Proof Assistant!

POPLMark Part A

Summary of the encoding techniques and tools used by the available submissions:

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		

Library vs special-purpose proof assistants?

Source: https://www.seas.upenn.edu/pclub/poplmark

De Bruijn Syntax

Manipulations in the lambda calculus are often troublesome because of the need for re-naming bound variables. For example, if a free variable in an expesssion has to be replaced by a second expression, the danger arises that some free variable of the second expression bears the same name as a bound variable in the first one, with the effect that binding is introduced where it is not intended. Another case of re-naming arises if we want to establish the equivalence of two expressions in those situations where the only difference lies in the names of the bound variables (i.e. when the equivalence is so-called α-equivalence).

In particular in machine-manipulated lambda calculus this re-naming activity involves a great deal of labour, both in machine time as in programming effort. It seems to be worth-while to try to get rid of the re-naming, or, rather, to get rid of names altogether.

Consider the following three criteria for a good notation:

(i) easy to write and easy to read for the human reader;
(ii) easy to handle in metalingual discussion;
(iii) easy for the computer and for the computer programmer.

The system we shall develop here is claimed to be good for (ii) and good for (iii). It is not claimed to be very good for (i); this means that for computer work we shall want automatic translation from one of the usual systems to our present system at the input stage, and backwards

Idea: α-equivalence = definitional equality

Binders are presented by **references** and induce a **scope change.**

Terms:

```
Inductive tm: Type :=
| var_tm : nat → tm
| app : tm → tm → tm
| lam : tm → tm.
```

Example term:

$\lambda\, x.\, z\ (\lambda\, y.\, (x\ y)\ z) \Rightarrow \lambda.\, 1\ (\lambda.\, (1\ 0)\ 2)$

```
lam (app (var_tm 1) (lam
(app (app (var_tm 1) (var_tm
0)) (var_tm 2))))
```

De Bruijn, Nicolaas Govert. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." 1972

(Parallel) Substitutions
de Bruijn ‘72

Goal: Instantiation with substitution, _ [_] : tm -> (ℕ -> tm) -> tm

$$(s \cdot \sigma)(x) := \begin{cases} s & \text{if } x = 0 \\ \sigma(x-1) & \text{otherwise} \end{cases}$$

$$\mathsf{id}(x) := x \qquad \uparrow(x) := x + 1$$

Primitives sufficient to define
e.g. beta-reduction

$$\begin{aligned} x[\sigma] &= \sigma(x) \\ (s\,t)[\sigma] &= (s[\sigma])\,(t[\sigma]) \\ (\lambda.\, s)[\sigma] &= \lambda.\, (s[\Uparrow\sigma]) \end{aligned} \qquad (\sigma \circ \tau)(x) = \sigma(x)[\tau]$$

$$\Uparrow\sigma := 0 \cdot (\sigma \circ \uparrow)$$

Requires again substitution

Substitution without copy and paste: Altenkirch, Burke, Wadler, LFMTP ‘25

Two-Level Approach: Adams, R.: Formalized metatheory with terms represented by an indexed family of types. Types for Proofs and Programs, ‘06.

Goal: Prove substitutivity of reduction, i.e. that $s \succ t$ implies $s[\sigma] \succ t[\sigma]$.

$$
\begin{aligned}
s[\sigma][t[\sigma]..] &= s[\mathsf{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle][t[\sigma] \cdot \mathsf{var}] \\
&= s[(\mathsf{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathsf{var})]] && \text{compositionality} \\
&= s[(\mathsf{var}\,0)[t[\sigma] \cdot \mathsf{var}] \cdot (\sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathsf{var})]] && \text{distributivity} \\
&= s[(\mathsf{var}\,0)[t[\sigma] \cdot \mathsf{var}] \cdot \sigma(\langle\uparrow\rangle \circ [t[\sigma] \cdot \mathsf{var}])] && \text{associativity} \\
&= s[(t[\sigma] \cdot \mathsf{var})\,0 \cdot (\sigma \circ [\uparrow (t[\sigma] \cdot \mathsf{var})])] && \text{compositionality} \\
&= s[t[\sigma] \cdot (\sigma \circ [\mathsf{var}])] && \cdot, \text{ interaction} \\
&= s[t[\sigma] \cdot \sigma] && \text{right identity} \\
&= s[t[\sigma] \cdot (\mathsf{var} \circ [\sigma])] && \text{left identity} \\
&= s[(t \cdot \mathsf{var}) \circ [\sigma]] && \text{distributivity} \\
&= s[t \cdot \mathsf{var}][\sigma]. && \text{compositionality}
\end{aligned}
$$

The Sigma Calculus [Abadi et al. ‘96]

A Convergent [Curien et al. ‘96] + Complete [Schäfer et al. ‘15] Rewriting System

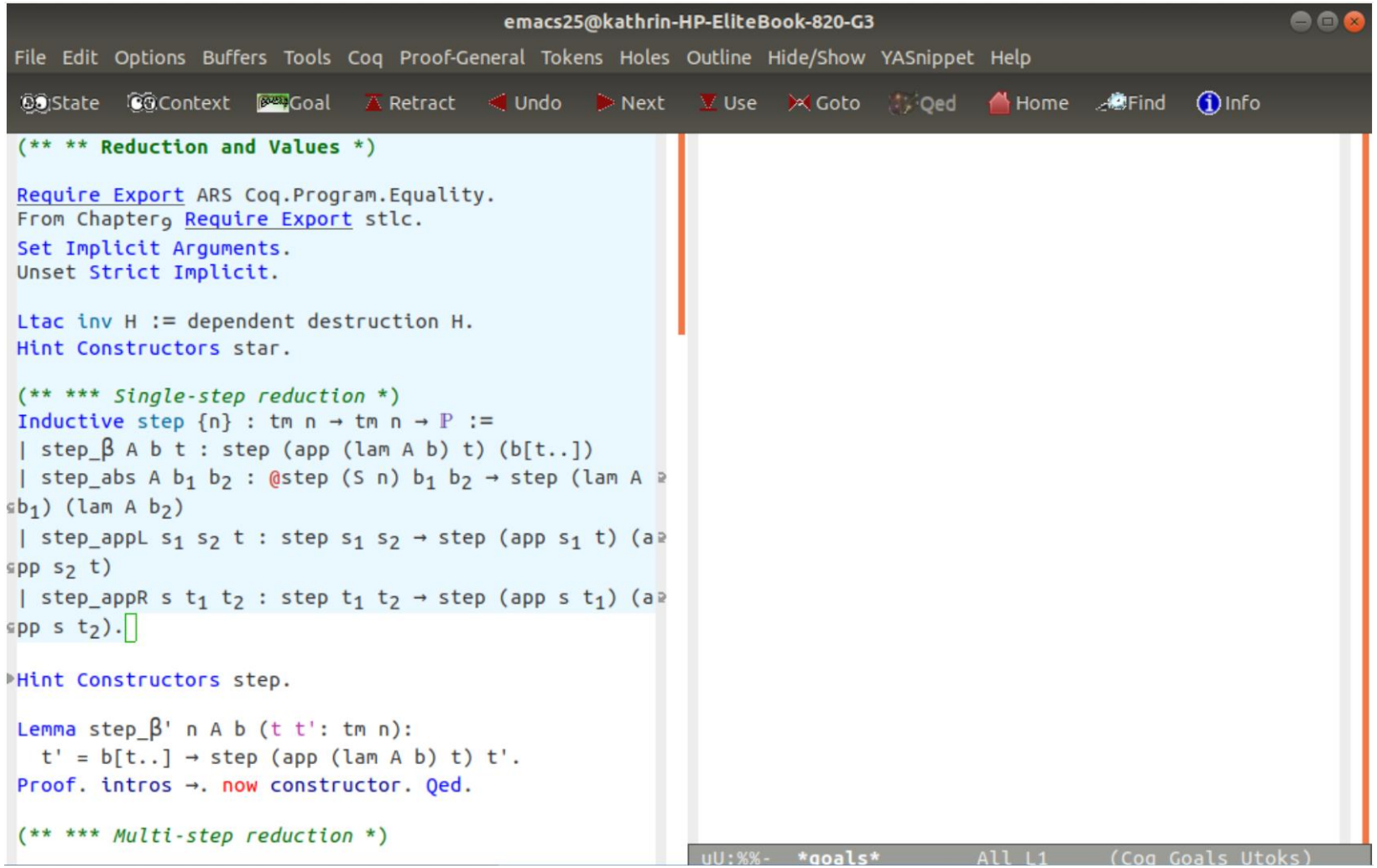
$$(st)[\sigma] \equiv (s[\sigma])(t[\sigma])$$
$$(\lambda.\, s)[\sigma] \equiv \lambda.\, (s[0 \cdot \sigma \circ \uparrow])$$
$$0[s \cdot \sigma] \equiv s$$
$$\uparrow \circ (s \cdot \sigma) \equiv \sigma$$
$$s[\mathsf{id}] \equiv s$$
$$0[\sigma] \cdot (\uparrow \circ \sigma) \equiv \sigma$$

$$\mathsf{id} \circ \sigma \equiv \sigma$$
$$\sigma \circ \mathsf{id} \equiv \sigma$$
$$(\sigma \circ \tau) \circ \theta \equiv \sigma \circ (\tau \circ \theta)$$
$$(s \cdot \sigma) \circ \tau \equiv s[\tau] \cdot \sigma \circ \tau$$
$$s[\sigma][\tau] \equiv s[\sigma \circ \tau]$$
$$0 \cdot \uparrow \equiv \mathsf{id}$$

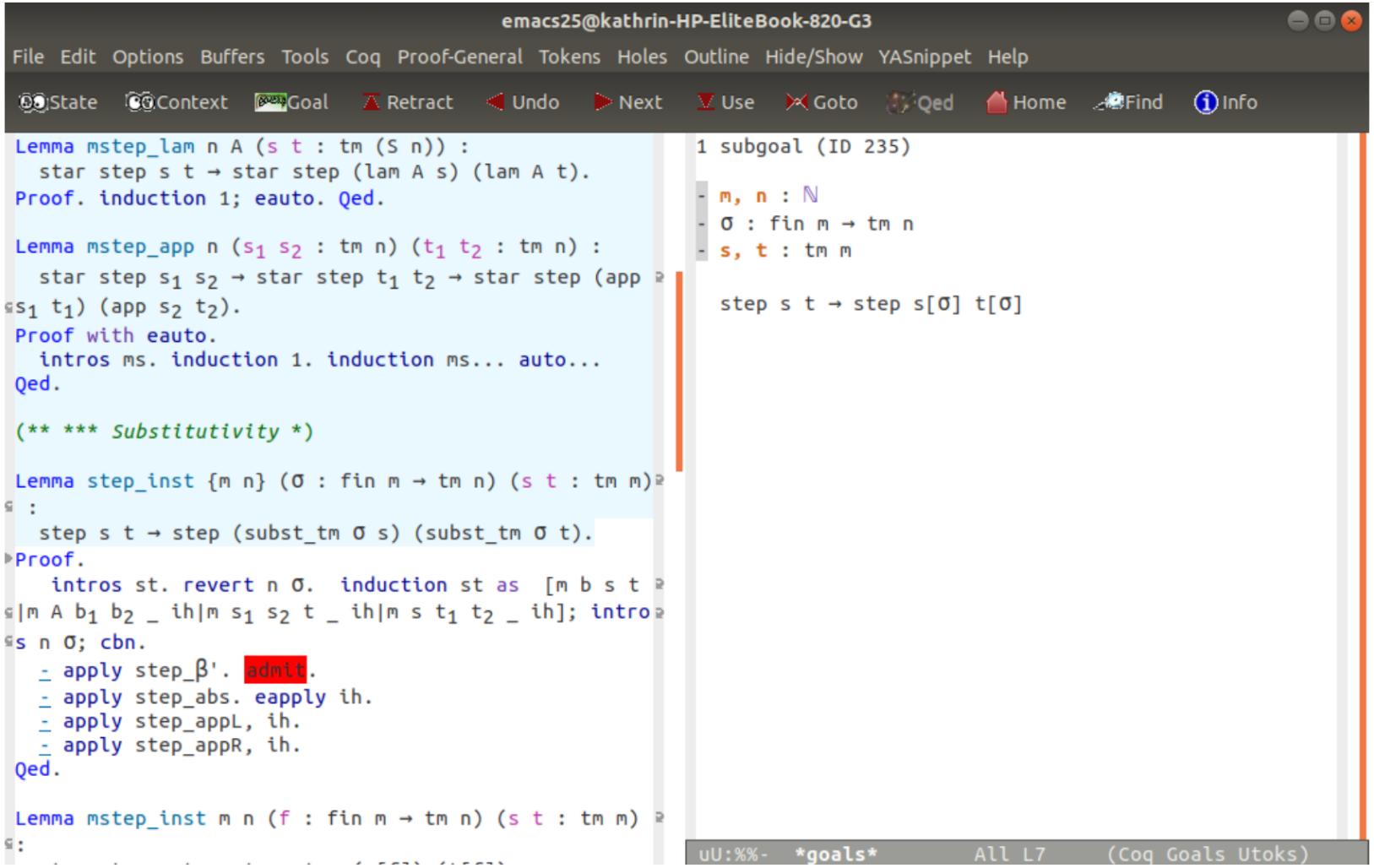
Implemented via a rewriting tactic called `asimpl` that normalizes a term using the rewriting system

$s = t$ can be decided via the above rewriting system

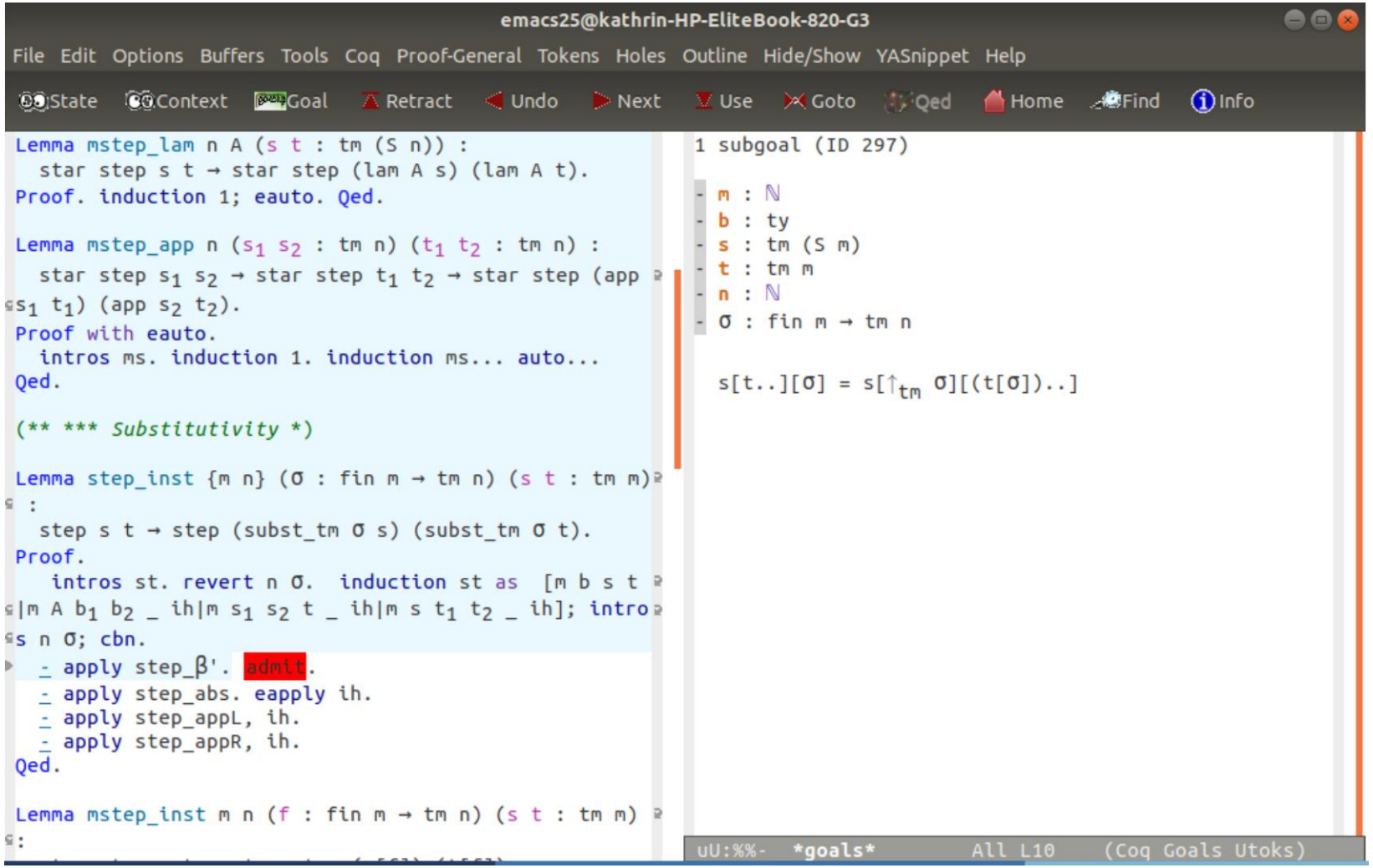
Demo: A Proof of Type Safety



Demo: A Proof of Type Safety



Demo: A Proof of Type Safety



Demo: A Proof of Type Safety

… and that's without the proofs!

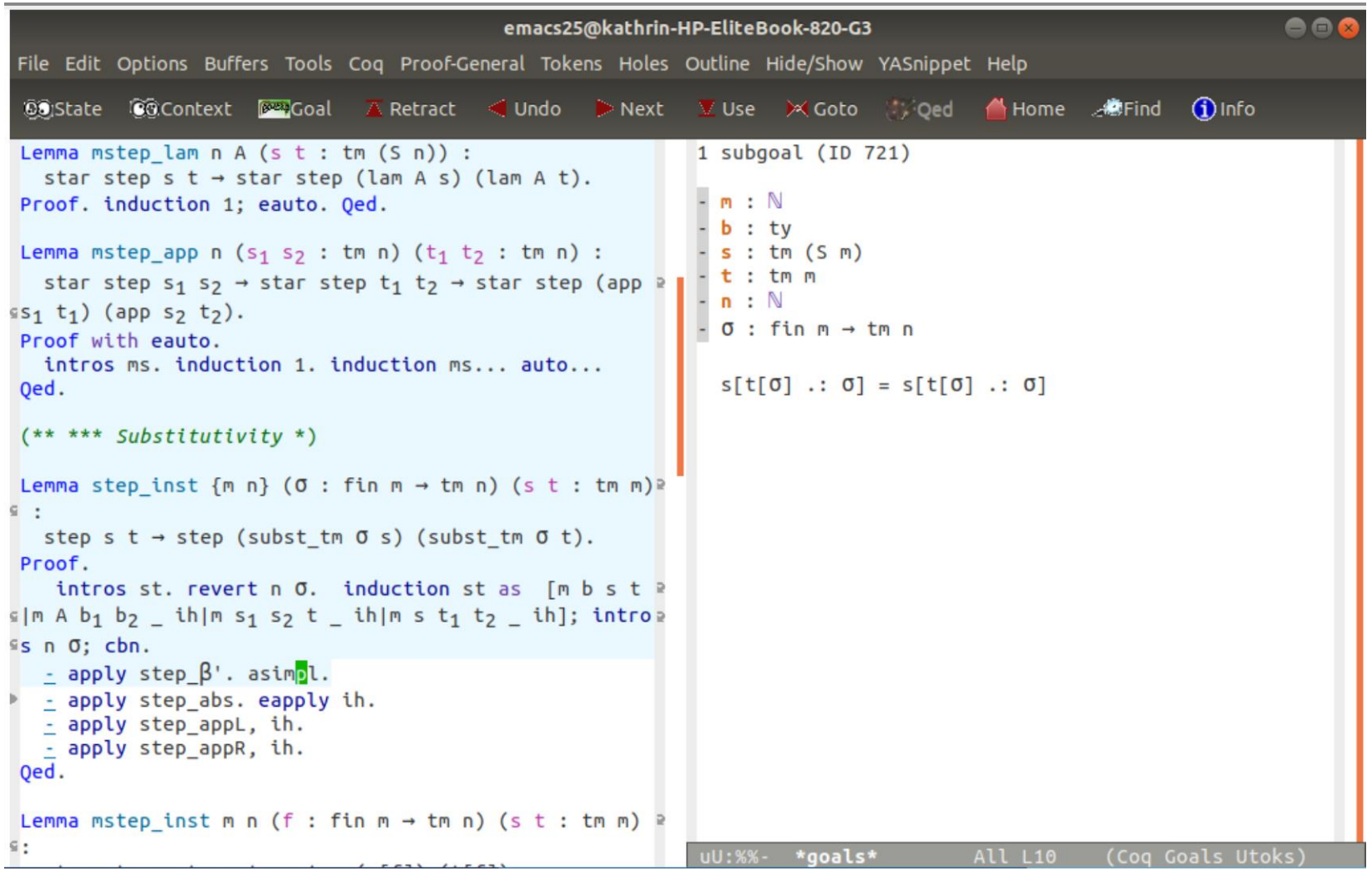
Goal: Prove substitutivity of reduction, i.e. that $s \succ t$ implies $s[\sigma] \succ t[\sigma]$.

$$
\begin{aligned}
s[\sigma][t[\sigma]..] &= s[\mathrm{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle][t[\sigma] \cdot \mathrm{var}] \\
&= s[(\mathrm{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathrm{var})]] && \text{compositionality} \\
&= s[(\mathrm{var}\,0)[t[\sigma] \cdot \mathrm{var}] \cdot (\sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathrm{var})]] && \text{distributivity} \\
&= s[(\mathrm{var}\,0)[t[\sigma] \cdot \mathrm{var}] \cdot \sigma(\langle\uparrow\rangle \circ [t[\sigma] \cdot \mathrm{var}])] && \text{associativity} \\
&= s[(t[\sigma] \cdot \mathrm{var})\,0 \cdot (\sigma \circ [\uparrow (t[\sigma] \cdot \mathrm{var})])] && \text{compositionality} \\
&= s[t[\sigma] \cdot (\sigma \circ [\mathrm{var}])] && \cdot,\ \text{interaction} \\
&= s[t[\sigma] \cdot \sigma] && \text{right identity} \\
&= s[t[\sigma] \cdot (\mathrm{var} \circ [\sigma])] && \text{left identity} \\
&= s[(t \cdot \mathrm{var}) \circ [\sigma]] && \text{distributivity} \\
&= s[t \cdot \mathrm{var}][\sigma]. && \text{compositionality}
\end{aligned}
$$

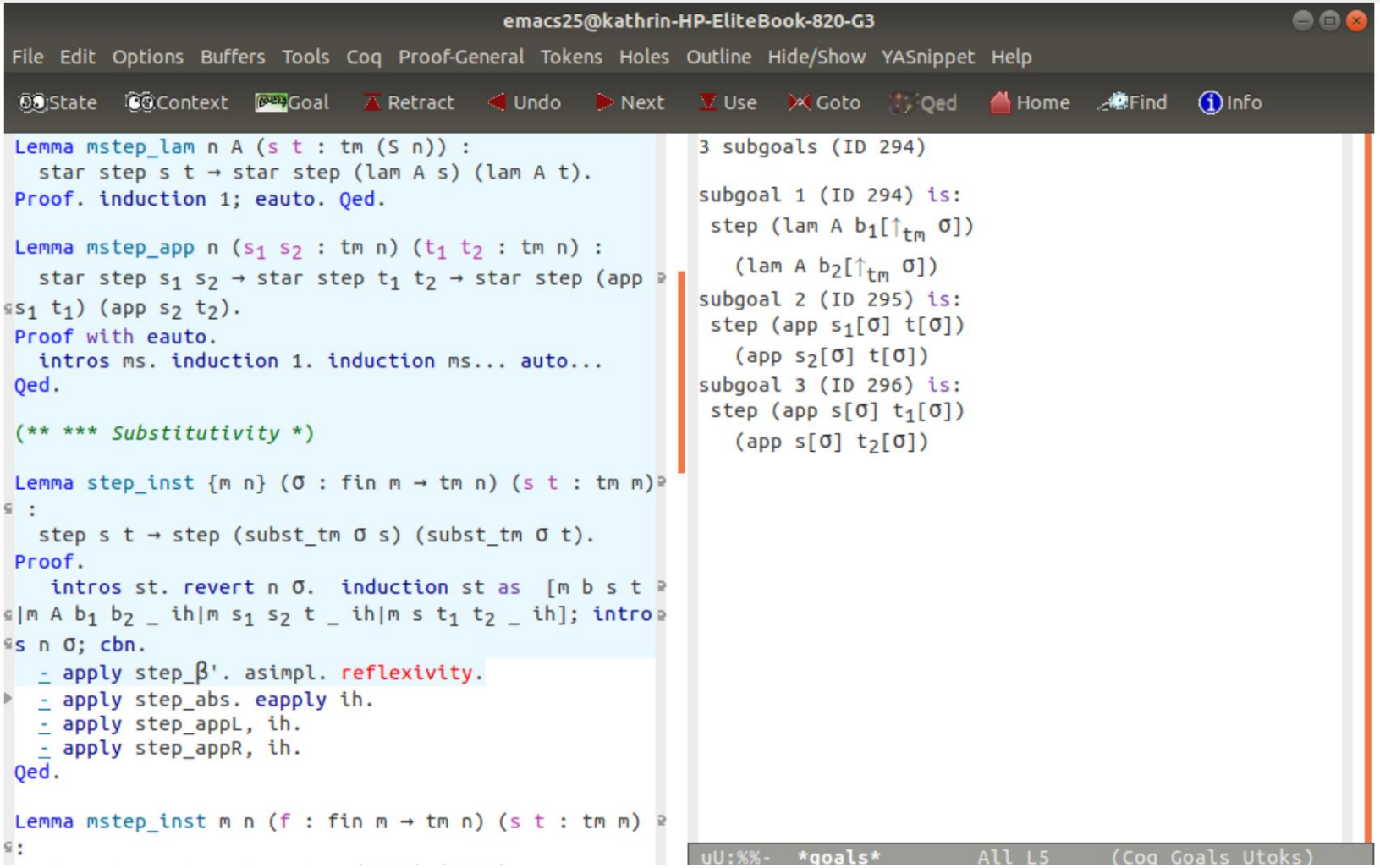
Which lemmas do we need?

How to apply them?

Demo: A Proof of Type Safety



Demo: A Proof of Type Safety



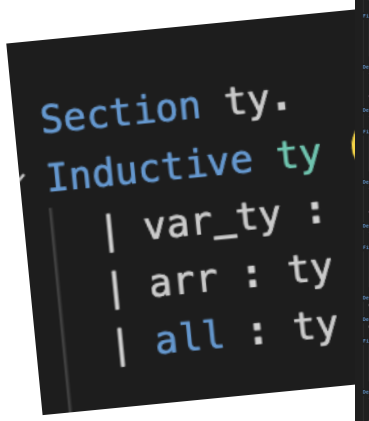
Autosubst

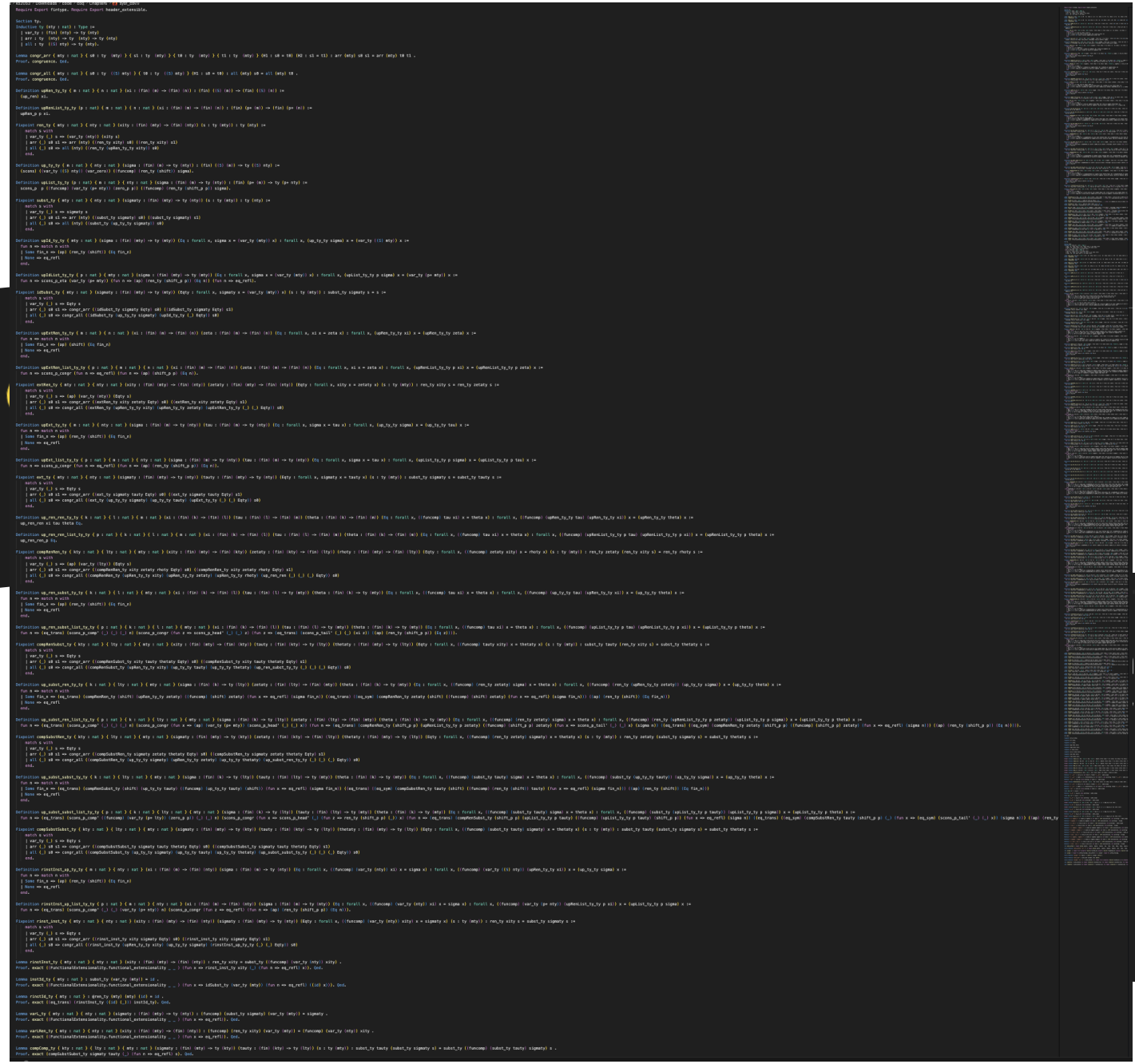
A **choice** of
- Binder representation
- Definition of substitutions
- Set of substitution lemmas

that requires minimum input from the user

Another popular concrete representation is de Bruijn's nameless representation. De Bruijn indices are easy to understand and support the full range of induction principles needed to reason over terms. [..] while the notation clutter is **manageable for "toy" examples** of the size of the simply-typed lambda calculus, we have found it becomes **quite a heavy burden even for fairly small languages like** $F_{<:}$.

[Aydemir et al. '05]


Section ty.
Inductive ty
| var_ty :
| arr : ty
| all : ty




(nty) (nvl)
) (nvl)
nty) (nvl)

One Main Culprit: Compositionality of Instantiation

Lemma 3.4 (Compositionality).

1. $(\Uparrow^* \xi) \circ (\Uparrow^* \zeta) \equiv \Uparrow^* (\xi \circ \zeta)$
2. $s\langle\xi\rangle\langle\zeta\rangle = s\langle\xi \circ \langle\zeta\rangle\rangle$
3. $(\Uparrow^* \xi) \circ (\Uparrow \tau) \equiv \Uparrow (\xi \circ \tau)$
4. $s\langle\xi\rangle[\tau] = s[\xi \circ \tau]$
5. $(\Uparrow \sigma) \circ (\Uparrow^* \zeta) \equiv \Uparrow (\sigma \circ \langle\zeta\rangle)$
6. $s[\sigma]\langle\zeta\rangle = s[\sigma \circ \langle\zeta\rangle]$
7. $(\Uparrow \sigma) \circ [\Uparrow \tau] \equiv \Uparrow (\sigma \circ [\tau])$
8. $s[\sigma][\tau] \equiv s[\sigma \circ [\tau]]$.

Autosubst

A **choice** of
- Binder representation
- Definition of substitutions
- Set of substitution lemmas

that requires minimum input from the user

+

A **library format** to automatically generate the corresponding boilerplate, including automation tactic
to require minimum input from the user

Different solutions:
Special-purpose proof assistants?
Code generation?
Universes of syntax with binding?

Autosubst Automation

.sig

Specification, e.g.

```
app : term → term → term
lam : (term → term) → term
```

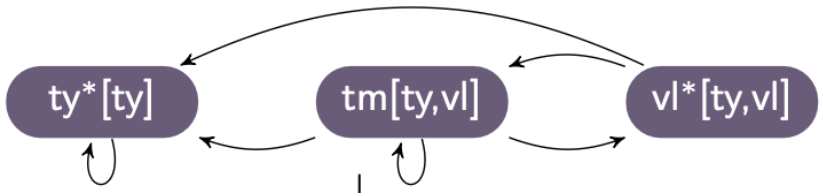
⇓ Autosubst 2 compiler

.v

Output file

Expressions
+ Instantiation _[_]
+ Reasoning

Automatically determined which substitutions are required



Schäfer, Tebbi, Smolka – CPP '15
S, Schäfer, Kaiser – CPP '19

Supported Syntax

```
ty, tm, vl : Type
arr : ty → ty → ty
all : (ty → ty) → ty

app : tm → tm → tm
tapp : tm → ty → tm
vt  : vl → tm

lam : ty → (vl → tm) → vl
tlam : (ty → tm) → vl
```

```
Inductive ty n : Type :=
| var_ty : fin n → ty n
| arr : ty n → ty n → ty n
| all : ty (n + 1) → ty n.

Inductive tm m n: Type :=
| app : tm m n → tm m n→ tm m n
| tapp : tm m n → ty m → tm m n
| vt : vl m n → tm m n
with vl m n : Type :=
| var_vl : fin n → vl m n
| lam : ty n → tm m (n + 1) → vl m n
| tlam : tm (m + 1) n → vl m n.
```

- Polyadic binders
- First-class renamings
- External sorts/sort constructors
- Many-sorted syntax
- Mutual inductive syntax
- Variadic syntax
- Simplified definitions for first-order sorts
- Modular syntax

Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions – S., Schäfer, Kaiser, CPP ‘19

Supporting Many-Sorted Syntax by Vector Substitutions

Example: Call-by-Value System F

Expressions

$$
\begin{array}{rcll}
A, B \in ty & ::= & X \mid A \to B \mid \forall X.A & \text{Types} \\
s, t \in tm & ::= & s\,t \mid s\,A \mid v & \text{Terms} \\
u, v \in vl & ::= & x \mid \lambda(x : A).s \mid \Lambda X.s & \text{Values}
\end{array}
$$

Substitutions *Vectorise* parallel substitutions:

$$\_[\_;\_] : \mathsf{vl} \to (\mathbb{N} \to \mathsf{ty}) \to (\mathbb{N} \to \mathsf{vl}) \to \mathsf{vl}$$

Reasoning Lift reasoning principles

Variadic Syntax

- **Variadic binders** bind a variadic number of n variables at once, e.g. in a **multivariate λ-calculus**:

$$s, t \in tm_k ::= \mathsf{var}\, x \mid \mathsf{app}\, s^k\, \{t_1^k .. t_n^k\} \mid \lambda_n.s^{n+k} \qquad\qquad x \in \mathsf{fin\, k}$$

$$\begin{array}{cccccccc} \mathsf{s} & & .. & 0 & 1 & 2 & & \\ & & & \mid & \mid & \mid & \ldots & \\ \lambda_n.s & 0 & .. & \mathsf{n} & 1+\mathsf{n} & 2+\mathsf{n} & & \end{array}$$

- Other examples: Pattern matching, recursive let-bindings

1. **Variadic shifting** $\uparrow^m$: $\mathsf{fin\, n} \to \mathsf{fin}\,(\mathsf{m}+\mathsf{n})$
2. **Variadic head**, $\mathsf{hd}_m : \mathsf{fin\, m} \to \mathsf{fin}\,(\mathsf{m}+\mathsf{n})$
3. **Variadic extension** $\_ \cdot_m \_ : (\mathsf{fin\, m} \to X) \to (\mathsf{fin\, n} \to X) \to (\mathsf{fin}\,(\mathsf{m}+\mathsf{n}) \to X)$, which precedes an arbitrary stream $\tau : \mathsf{fin\, n} \to X$ with a new stream $\sigma : \mathsf{fin\, m} \to X$:

$+$ definition of instantiation $+$ adaption of reasoning principles

Different Versions of the Autosubst Library

- Autosubst
  - Implemented using Ltac

```
Inductive term :=
| Var (x : var)
| App (s t : term)
| Lam (s : {bind term}).
```

- Autosubst 2
  - Re-implementation in Haskell [S, Schäfer, Kaiser, ‘19]
  - OCaml version [Dapprich ‘21]
  - MetaCoq version [Dapprich ‘21]

- Lean emulation by Marmaduke, A., Ingle, A., & Morris, J. G. (2025)

Some trade-offs:

- How expressive?
- What can be generated? Tactics, notations?
- How maintainable for different versions of Rocq?
- Can the code be inspected/changed manually?

Projects using Autosubst/Autosubst 2

- Adjedi et al. use Autosubst 2 to deal with the raw syntax of a mechanization of the metatheory of Martin-Löf Type Theory.
- Castro shows that bounded quantification makes subtyping and type checking undecidable in System $F_{\leq:}$ using the synthetic approach of the Coq Library of Undecidability Proofs
- Dudenhefner and Pautasso use Autosubst to provide binder boilerplate for a purely syntactical proof of strong normalization for the simply typed λ-calculus
- Forster et al. present a formalization of the metatheory of call-by-push-value
- Forster, Kirst and Wehr prove completeness theorems for first-order logic
- Giarrusso et al. use Autosubst to generate metatheory of Scala's core type system Đ
- Kaiser et al. formally verify the correspondence between the two sorted presentation of System F and its presentation as a pure type system
- Mizuno and Sumii formally verify the correspondence between call-by-need and call-by-name. Their development also includes a proof of the standardization theorem for λ-calculus.
- Pottier presents a machine checked development of the CPS translation for the pure λ-calculus with a let construct.
- Spies and Forster formalise undecidability results concerning higher-order unification in the simply-typed lambda-calculus with beta-conversion
- Timany et al. present a logical relations model of a higher-order functional programming language with impredicative and imperative features and show that scoped effectful computations are observationally pure.
- Timany and Birkedal build a tool for interactive mechanized relational verification of programs written in a concurrent higher-order imperative programming language with continuations.
- Tirore et al. use Autosubst 2 to generate syntax for multiparty session types
- Wand et al. present a complete reasoning principle for contextual equivalence in an untyped probabilistic programming language.
- Winterhalter uses Autosubst 2 to deal with binders for ghost type theory (GTT)

… but: also complicates things.

Well Scoped Syntax

Idea: Terms are indexed by the upper bound of free variables.
For example:

```
Inductive tm (n : nat) :=
| var : fin n → tm n
| app : tm n → tm n → tm n
| lam : tm (S n) → tm n.
```

Example: η-reduction rule for λ-terms

$$\frac{x \notin \mathrm{FV}\ s}{\lambda x.\ s\ x \rhd s} \qquad \frac{}{\lambda(s[\uparrow]\ 0) \rhd s}$$

Well scoped by construction – if
the shift is not included, an error is thrown

Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs, ‘06.
Bird, R., Paterson, R.: de Bruijn notation as a nested datatype. JFP, ‘99.

One Bottleneck: Efficiency of Rewriting

Mathis Bouverot-Dupuis:
Recent internship with Théo Winterhalter on a reflective version of the `asimpl` tactic.

What Other Binder Support Should There Be?

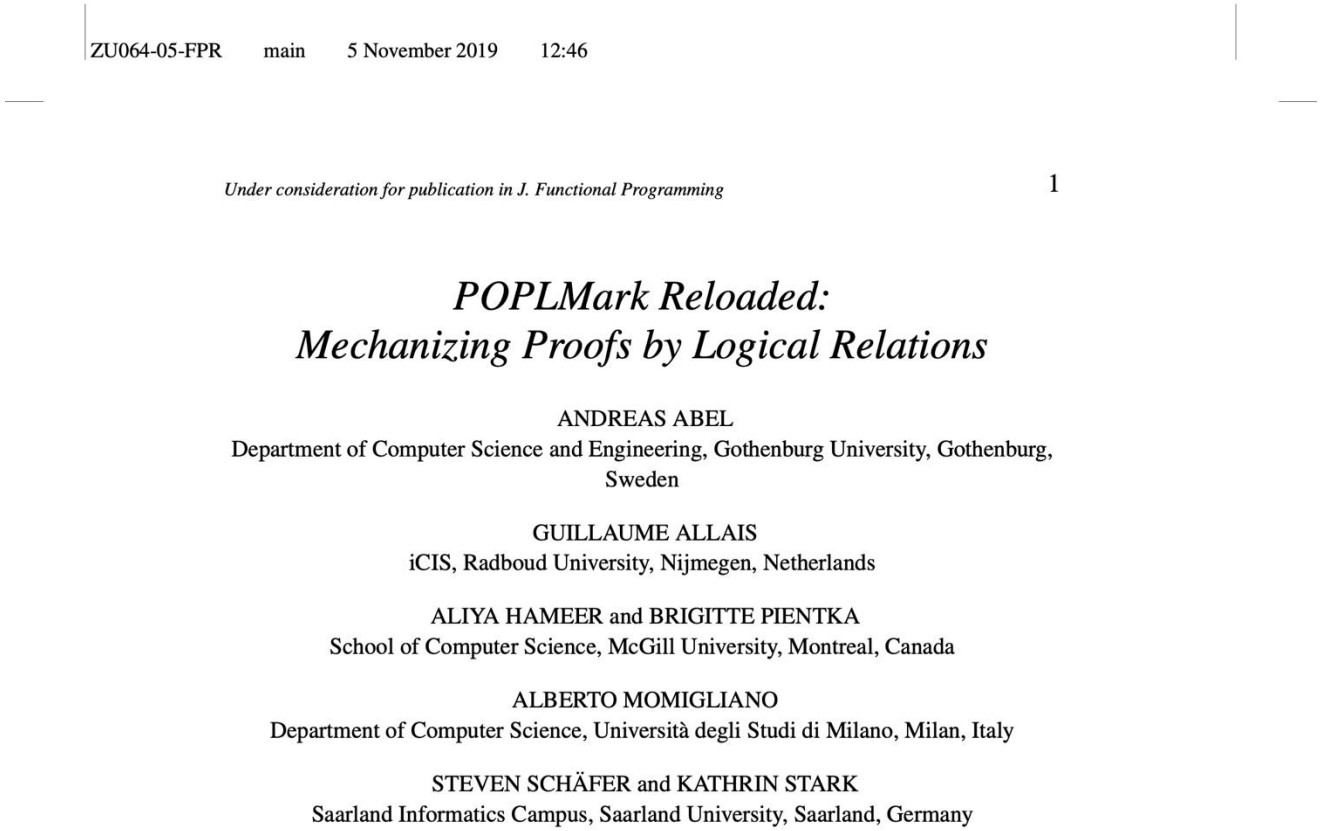
- Support for not only decidability but matching with assumptions
- Support for traversals over syntax with binders
  - Allais et al., JFP ’21 in Agda
- Intrinsically-encoded inductive types, e.g.
  - Allais et al., JFP ’21
  - Fiore/ Szamozvancev, POPL ‘22
- Binding for Substructural Languages, e.g.
  - Wood and Atkey - A framework for semiring-annotated type systems, ESOP ‘22
  - Zackon, Sano, Momigliano, Pientka – Split Decisions: Explicit Contexts for Substructural Languages, CPP ’25

What Form Should a Binder Library Take?

Comparison of three approaches:

- Special-purpose proof assistant (here: Beluga)
- Library with code generation (Autosubst)
- Library using an intrinsic representation by Allais et al.

How many/which theoretical foundations are necessary?



ZU064-05-FPR main 5 November 2019 12:46

Under consideration for publication in J. Functional Programming 1

POPLMark Reloaded:
Mechanizing Proofs by Logical Relations

ANDREAS ABEL
Department of Computer Science and Engineering, Gothenburg University, Gothenburg, Sweden

GUILLAUME ALLAIS
iCIS, Radboud University, Nijmegen, Netherlands

ALIYA HAMEER and BRIGITTE PIENTKA
School of Computer Science, McGill University, Montreal, Canada

ALBERTO MOMIGLIANO
Department of Computer Science, Università degli Studi di Milano, Milan, Italy

STEVEN SCHÄFER and KATHRIN STARK
Saarland Informatics Campus, Saarland University, Saarland, Germany

Today’s Talk

1. **Autosubst – a library for binders**
2. Coq à la Carte - Towards a library for modular proofs

Today’s Talk

1. Autosubst – a library for binders
2. **Coq à la Carte - Towards a library for modular proofs**

Proof libraries are useful…
but how do they cope with
inductive proofs?

Extending Inductive Proofs

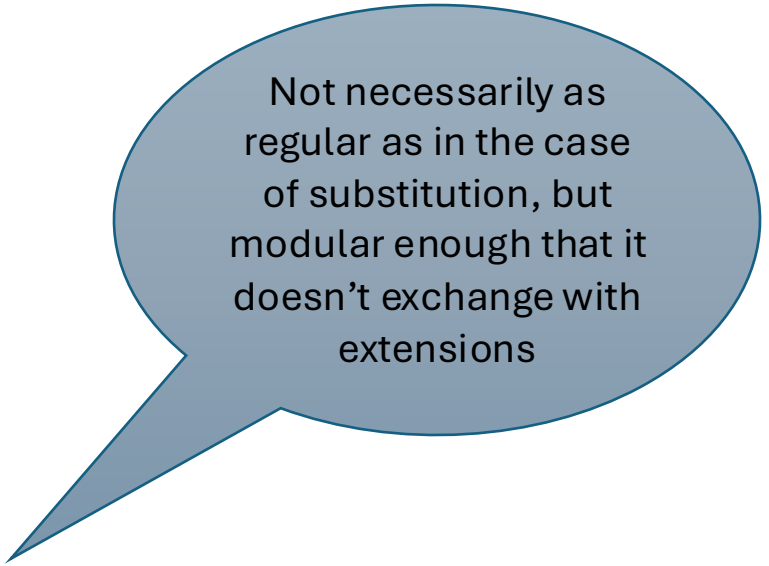
- You start with the lambda-calculus:

$$s, t \in tm ::= x \mid s\ t \mid \lambda x.\, s$$

  - You give
    - Recursive functions on terms
    - Proofs by induction on terms
    - And predicates and proofs over terms
- … and then want to **extend** this calculus, e.g. by Boolean expressions:

$$s, t \in tm ::= \cdots \mid b \mid if\ s\ then\ t\ else\ u$$

- **True modularity:** "[..] add new cases to the datatype [..] without recompiling existing code."



Can we avoid the copy-paste?

Related Work (I)

True Modularity in Haskell: Data Types a la Carte [Swierstra, 2008]

- Features as functors, e.g.

```
Inductive exp_λ (exp : Type) :=
| var : nat → exp λ exp
| app : exp → exp → exp λ exp
| abs : exp → exp λ exp.
```

- A general expression type as fixed point of functors:

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

  and variants which instantiate the general data type with coproducts of feature functors.

- Functions = algebras, assembling via type classes

Related Work (II)

True Modularity in a Proof Assistant?

- **Problem:** The general expression type is impossible in a proof assistant due to the restriction to positivity!

- Solution: **Encode the functor.**
  - Modular Type Safety Proofs in Agda [Schwaab and Siek, 2013]
  - Meta-Theory a la Carte [Delaware et al., 2008]
  - Generic Data Types a la Carte [Keuchel et al., 2013]
  - Modular Monadic Meta-Theory [Delaware et al., 2013]

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.* 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

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).

A Practical Approach to Modular Syntax

- Modular syntax via functions and variants with direct injections inspired by Data Types a la Carte [Swierstra '08]

```
Inductive exp_λ (exp : Type) :=
  | var : nat → exp_λ exp
  | app : exp → exp → exp_λ exp
  | abs : exp → exp_λ exp.
```

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

⟹

```
Inductive exp :=
  | inj_λ : exp_λ exp → exp
  | inj_𝔹 : exp → exp.
```

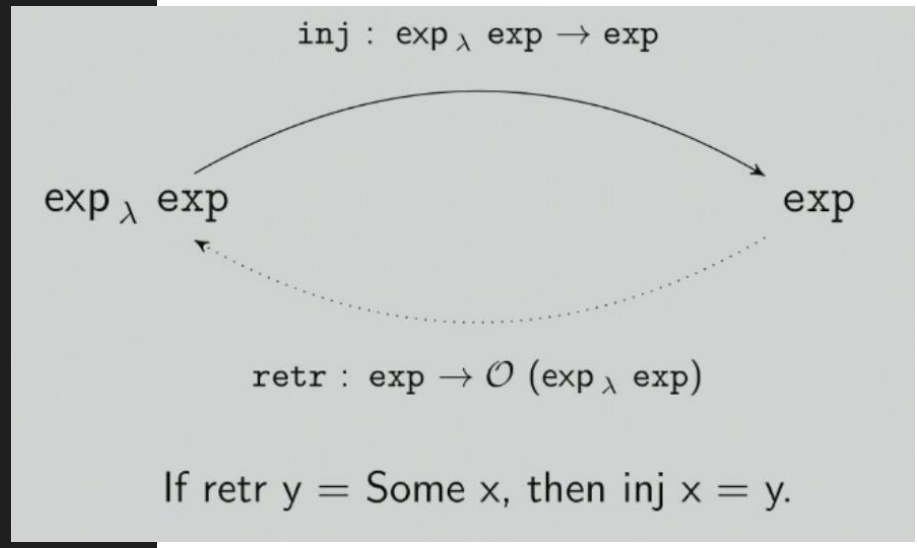
- **Tool support:**
  - Boilerplate generation with an extension of Autosubst 2
  - Assembling via MetaRocq [Sozeau et al., '19]
- **Result:**
  - Practical modular developments
  - Improvement from 1000 loc/feature to 125 loc/feature

Demo

Generated Parts: Defining a New Variant

```
Inductive exp_2 :=
| inj_var  : exp_var  exp_2 → exp_2
| inj_λ    : exp_λ    exp_2 → exp_2
| inj_𝔹    : exp_𝔹    exp_2 → exp_2.
```

```
Section exp_plus.
Variable exp : Type.

Inductive exp_plus  : Type :=
  | atom : nat   -> exp_plus
  | plus : exp   -> exp   -> exp_plus.

Variable retract_exp_plus : retract exp_plus exp.

Definition atom_  (s0 : nat ) : exp :=
  inj (atom s0).

Definition plus_  (s0 : exp ) (s1 : exp ) : exp :=
  inj (plus s0 s1).

End exp_plus.
```



Possibility to lift constructors from features to variants [Swierstra, ‘08]

Demo

Generated Parts

```
Section exp_opt.
Variable exp : Type.

Inductive exp_opt  : Type :=
  | none : exp_opt
  | some : exp   -> exp_opt .

Variable retract_exp_opt : retract exp_opt exp.

Definition none_  : _ :=
  inj (none ).

Definition some_  (s0 : exp ) : _ :=
  inj (some s0).

End exp_opt.
```

Demo

Generated Parts

```
Section exp.
Inductive exp  : Type :=
  | In_exp_plus : exp_plus exp  -> exp
  | In_exp_arr : exp_arr exp  -> exp
  | In_exp_opt : exp_opt exp  -> exp .

Lemma congr_In_exp_plus  { s0 : exp_plus exp  } { t0 : exp_plus exp  } : s0 = t0 -> In_exp_plus s0
Proof. congruence. Qed.

Lemma congr_In_exp_arr  { s0 : exp_arr exp  } { t0 : exp_arr exp  } : s0 = t0 -> In_exp_arr s0 = I
Proof. congruence. Qed.

Lemma congr_In_exp_opt  { s0 : exp_opt exp  } { t0 : exp_opt exp  } : s0 = t0 -> In_exp_opt s0 = I
Proof. congruence. Qed.

Global Instance retract_exp_plus_exp  : retract (exp_plus exp) exp := Build_retract In_exp_plus (f
| In_exp_plus s => Datatypes.Some s
| _ => Datatypes.None
end) (fun s => eq_refl) (fun s t => match t with
| In_exp_plus t' => fun H => congr_In_exp_plus (eq_sym (Some_inj H))
| _ => some_none_explosion
end) .

Global Instance retract_exp_opt_exp  : retract (exp_opt exp) exp := Build_retract In_exp_opt (fun
| In_exp_opt s => Datatypes.Some s
| _ => Datatypes.None
end) (fun s => eq_refl) (fun s t => match t with
| In_exp_opt t' => fun H => congr_In_exp_opt (eq_sym (Some_inj H))
| _ => some_none_explosion
end) .

Global Instance retract_exp_arr_exp  : retract (exp_arr exp) exp := Build_retract In_exp_arr (fun
```

Generated proof of the retract property

Demo

User-defined Parts: Generation of Inductive Predicates

```
Variable exp: Type.
Context `{retract (exp_plus exp) exp}.

Variable wtyped : exp -> ty -> Prop.

Inductive wtyped_plus : exp -> ty -> Prop :=
| ok_value n : wtyped_plus (atom_ n) TNat
| ok_plus s t : wtyped s TNat -> wtyped t TNat -> wtyped_plus (plus_ s t) TNat.

Variable eval : exp -> exp -> Prop.

Inductive eval_plus : exp -> exp -> Prop :=
| stepl e1 e1' e2 : eval e1 e1' -> eval_plus (plus_ e1 e2) (plus_ e1' e2)
| stepr e1 e2 e2': eval e2 e2' -> eval_plus  (plus_ e1 e2) (plus_ e1 e2')
| sum n1 n2 : eval_plus  (plus_ (atom_ n1) (atom_ n2)) (atom_ (n1 + n2)).

Hint Constructors wtyped_plus eval_plus.
```

Demo

User-defined Parts: A Proof of Type Safety (Composition)

```
(** ** Composed development *)

Inductive wtyped : exp -> ty -> Prop :=
| wtyped_in_plus s A : wtyped_plus wtyped s A -> wtyped s A
| wtyped_in_arr s A : wtyped_arr wtyped s A -> wtyped s A
| wtyped_in_opt s A : wtyped_opt wtyped s A -> wtyped s A.

Hint Constructors wtyped_plus wtyped_arr wtyped_opt wtyped.

Inductive eval : exp -> exp -> Prop :=
| eval_in_plus s t : eval_plus eval s t -> eval s t
| eval_in_arr s t : eval_arr eval s t -> eval s t
| eval_in_opt s t : eval_opt eval s t -> eval s t.

Hint Constructors eval_plus eval_arr eval_opt eval.

Lemma wtyped_inv_plus :
  forall (s0 : exp_plus exp) (A : ty), wtyped (inj s0) A
    -> wtyped_plus wtyped (inj s0) A.
Proof.
  intros. inv H; inv H0; econstructor; eauto.
Qed.
Hint Resolve wtyped_inv_plus.
```

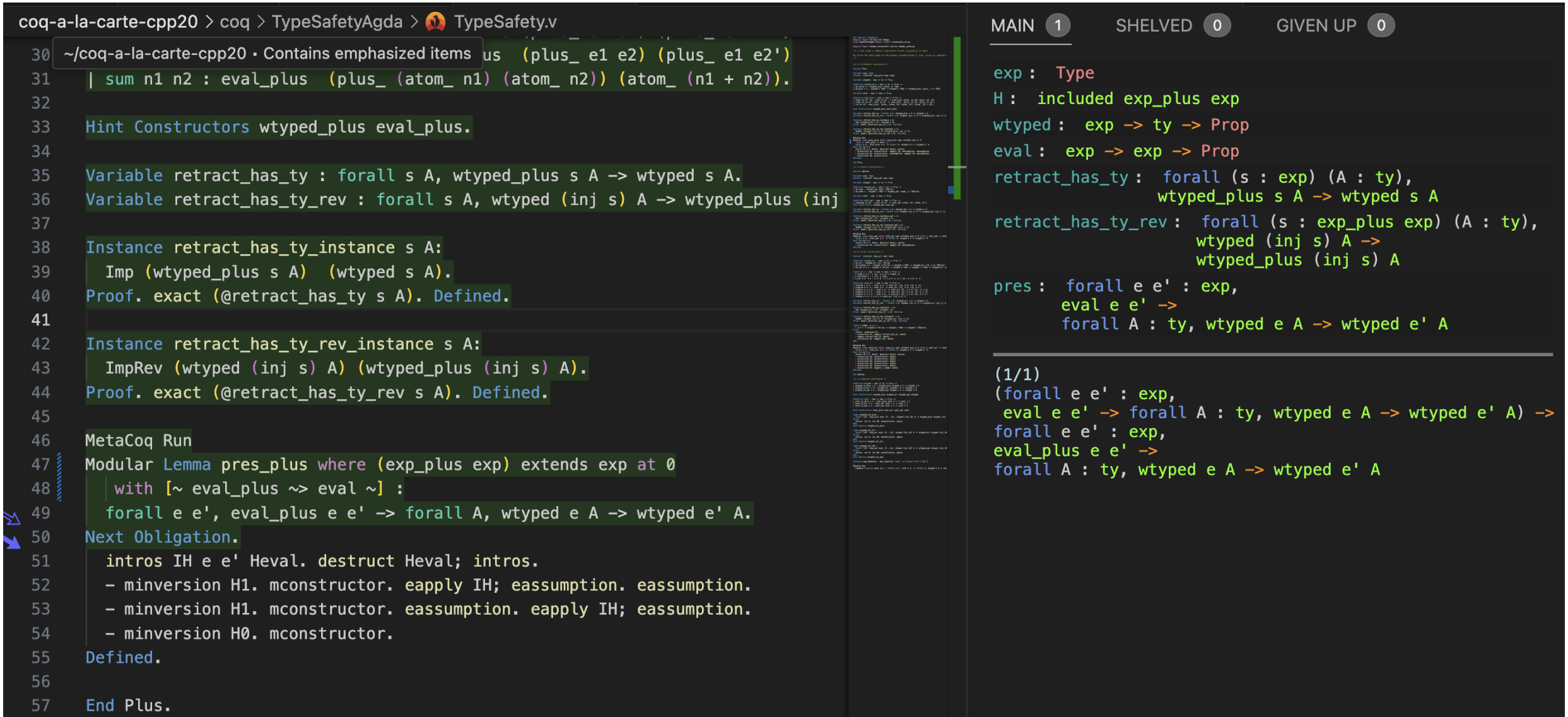
Demo

User-defined Parts: Generation of Inductive Predicates

```
Variable retract_has_ty : forall s A, wtyped_plus s A -> wtyped s A.
Variable retract_has_ty_rev : forall s A, wtyped (inj s) A -> wtyped_plus (inj s) A.

Instance retract_has_ty_instance s A:
  Imp (wtyped_plus s A)  (wtyped s A).
Proof. exact (@retract_has_ty s A). Defined.

Instance retract_has_ty_rev_instance s A:
  ImpRev (wtyped (inj s) A) (wtyped_plus (inj s) A).
Proof. exact (@retract_has_ty_rev s A). Defined.
```

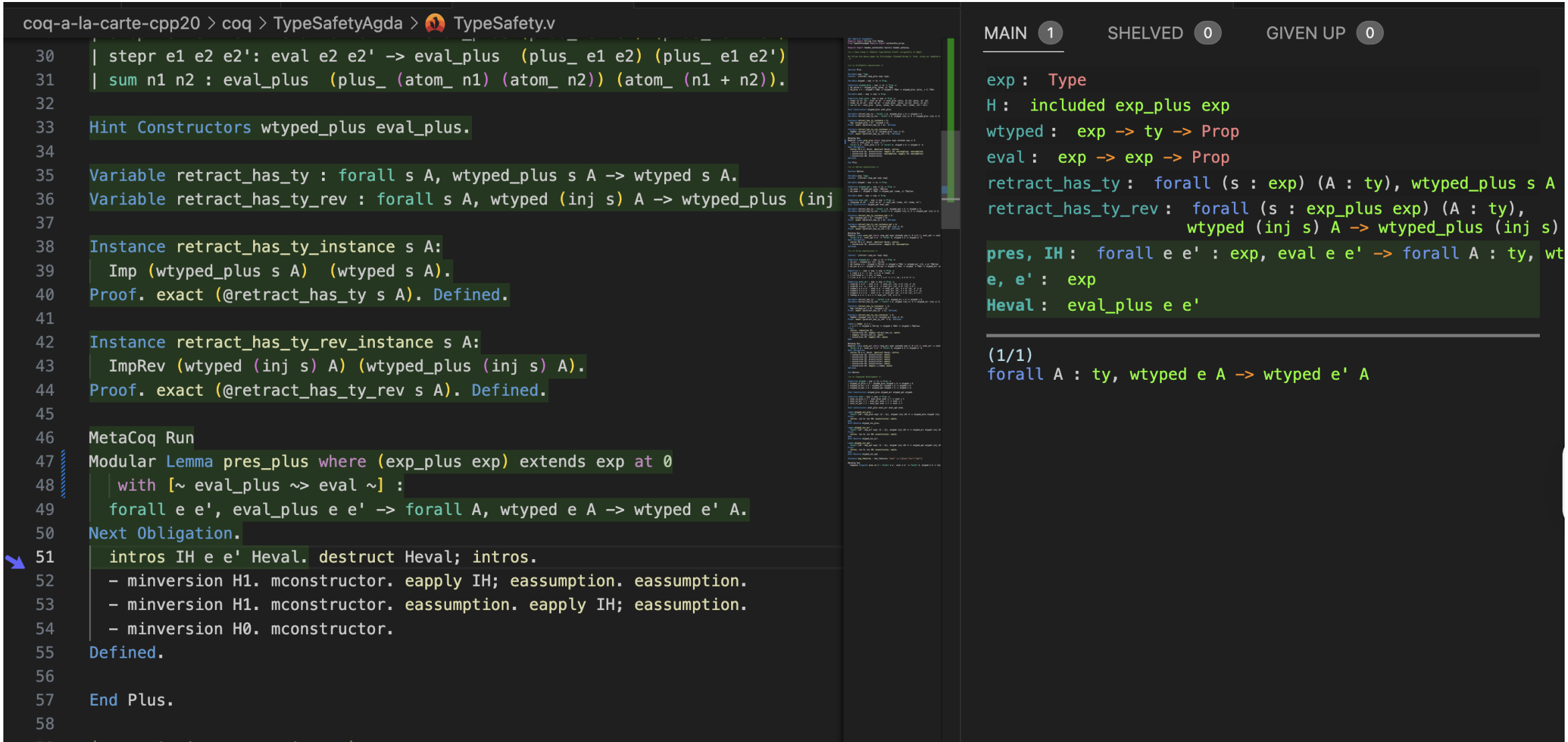
Demo

User-defined Parts: A Proof of Type Safety



Demo

User-defined Parts: A Proof of Type Safety



Demo

User-defined Parts: A Proof of Type Safety

coq-a-la-carte-cpp20 > coq > TypeSafetyAgda > TypeSafety.v

```
30  | stepr e1 e2 e2': eval e2 e2' -> eval_plus  (plus_ e1 e2) (plus_ e1 e2')
31  | sum n1 n2 : eval_plus  (plus_ (atom_ n1) (atom_ n2)) (atom_ (n1 + n2)).
32
33  Hint Constructors wtyped_plus eval_plus.
34
35  Variable retract_has_ty : forall s A, wtyped_plus s A -> wtyped s A.
36  Variable retract_has_ty_rev : forall s A, wtyped (inj s) A -> wtyped_plus (inj
37
38  Instance retract_has_ty_instance s A:
39    Imp (wtyped_plus s A)  (wtyped s A).
40  Proof. exact (@retract_has_ty s A). Defined.
41
42  Instance retract_has_ty_rev_instance s A:
43    ImpRev (wtyped (inj s) A) (wtyped_plus (inj s) A).
44  Proof. exact (@retract_has_ty_rev s A). Defined.
45
46  MetaCoq Run
47  Modular Lemma pres_plus where (exp_plus exp) extends exp at 0
48     with [~ eval_plus ~> eval ~] :
49    forall e e', eval_plus e e' -> forall A, wtyped e A -> wtyped e' A.
50  Next Obligation.
51    intros IH e e' Heval. destruct Heval; intros.
52    - minversion H1. mconstructor. eapply IH; eassumption. eassumption.
53    - minversion H1. mconstructor. eassumption. eapply IH; eassumption.
54    - minversion H0. mconstructor.
55  Defined.
56
57  End Plus.
58
59  (** ** Option expressions *)
```

MAIN 3 SHELVED 0 GIVEN UP 0

```
exp :  Type
H :  included exp_plus exp
wtyped :  exp -> ty -> Prop
eval :  exp -> exp -> Prop
retract_has_ty :  forall (s : exp) (A : ty),
                  wtyped_plus s A -> wtyped s A
retract_has_ty_rev :  forall (s : exp_plus exp) (A : ty),
                      wtyped (inj s) A ->
                      wtyped_plus (inj s) A
```

```
pres,
IH :  forall e e' : exp,
      eval e e' ->
      forall A : ty, wtyped e A -> wtyped e' A
e1, e1', e2 :  exp
H0 :  eval e1 e1'
A :  ty
H1 :  wtyped (plus_ e1 e2) A
```

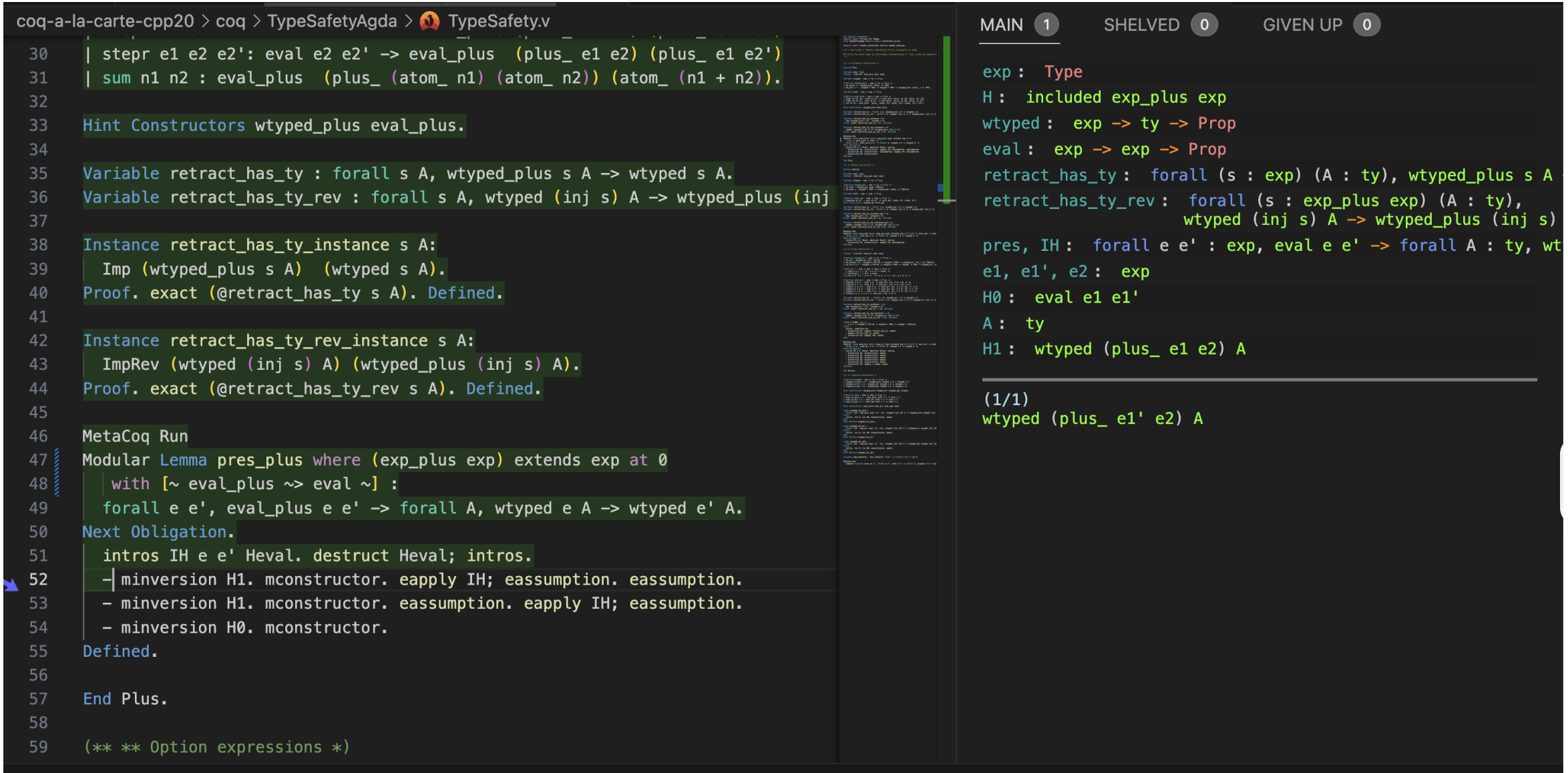
```
(1/3)
wtyped (plus_ e1' e2) A
```

```
(2/3)
wtyped (plus_ e1 e2') A
```

```
(3/3)
wtyped (atom_ (n1 + n2)) A
```

Demo

User-defined Parts: A Proof of Type Safety



Demo

User-defined Parts: A Proof of Type Safety

Tactic that automatically uses retracts to do case analysis on a modular data type

coq-a-la-carte-cpp20 > coq > TypeSafetyAgda > TypeSafety.v

```
30  | stepr e1 e2 e2': eval e2 e2' -> eval_plus  (plus_ e1 e2) (plus_ e1 e2')
31  | sum n1 n2 : eval_plus  (plus_ (atom_ n1) (atom_ n2)) (atom_ (n1 + n2)).
32
33  Hint Constructors wtyped_plus eval_plus.
34
35  Variable retract_has_ty : forall s A, wtyped_plus s A -> wtyped s A.
36  Variable retract_has_ty_rev : forall s A, wtyped (inj s) A -> wtyped_plus (inj
37
38  Instance retract_has_ty_instance s A:
39    Imp (wtyped_plus s A)  (wtyped s A).
40  Proof. exact (@retract_has_ty s A). Defined.
41
42  Instance retract_has_ty_rev_instance s A:
43    ImpRev (wtyped (inj s) A) (wtyped_plus (inj s) A).
44  Proof. exact (@retract_has_ty_rev s A). Defined.
45
46  MetaCoq Run
47  Modular Lemma pres_plus where (exp_plus exp) extends exp at 0
48      with [~ eval_plus ~> eval ~] :
49    forall e e', eval_plus e e' -> forall A, wtyped e A -> wtyped e' A.
50  Next Obligation.
51    intros IH e e' Heval. destruct Heval; intros.
52    - minversion H1. mconstructor. eapply IH; eassumption. eassumption.
53    - minversion H1. mconstructor. eassumption. eapply IH; eassumption.
54    - minversion H0. mconstructor.
55  Defined.
56
57  End Plus.
58
59  (** ** Option expressions *)
```

MAIN 1 SHELVED 0 GIVEN UP 0

```
exp :  Type
H :  included exp_plus exp
wtyped :  exp -> ty -> Prop
eval :  exp -> exp -> Prop
retract_has_ty :  forall (s : exp) (A : ty),
                  wtyped_plus s A -> wtyped s A
retract_has_ty_rev :  forall (s : exp_plus exp) (A : ty),
                      wtyped (inj s) A ->
                      wtyped_plus (inj s) A
pres,
IH :  forall e e' : exp,
      eval e e' ->
      forall A : ty, wtyped e A -> wtyped e' A
e1, e1', e2 :  exp
H0 :  eval e1 e1'
H1 :  wtyped_plus (inj (plus exp e1 e2)) TNat
H3 :  wtyped e1 TNat
H4 :  wtyped e2 TNat
```

```
(1/1)
wtyped (plus_ e1' e2) TNat
```

Demo

User-defined Parts: A Proof of Type Safety

Tactic that automatically uses retracts to build instances of a modular type

coq-a-la-carte-cpp20 > coq > TypeSafetyAgda > TypeSafety.v

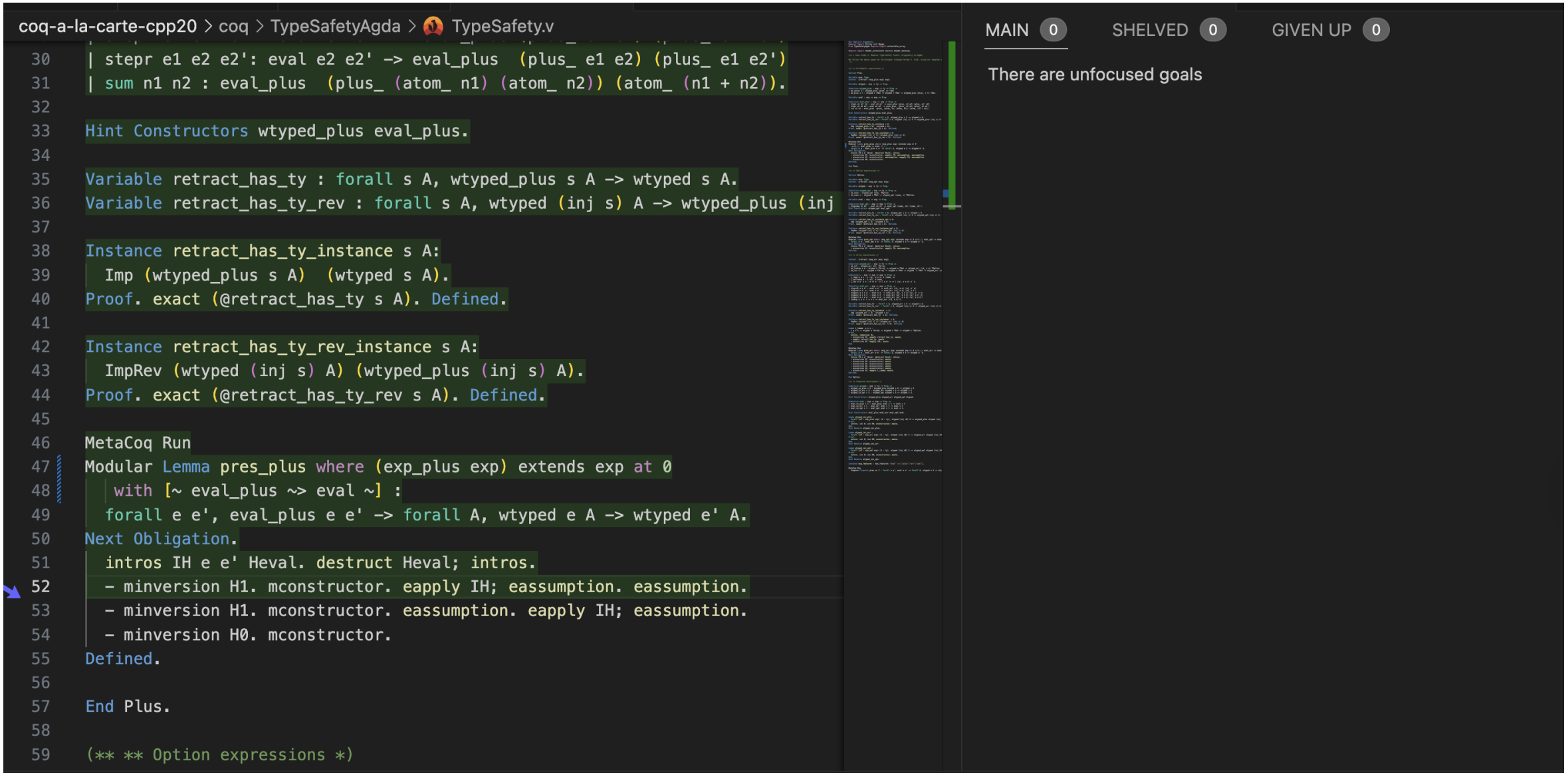
```
30  | stepr e1 e2 e2': eval e2 e2' -> eval_plus  (plus_ e1 e2) (plus_ e1 e2')
31  | sum n1 n2 : eval_plus  (plus_ (atom_ n1) (atom_ n2)) (atom_ (n1 + n2)).
32
33  Hint Constructors wtyped_plus eval_plus.
34
35  Variable retract_has_ty : forall s A, wtyped_plus s A -> wtyped s A.
36  Variable retract_has_ty_rev : forall s A, wtyped (inj s) A -> wtyped_plus (inj
37
38  Instance retract_has_ty_instance s A:
39    Imp (wtyped_plus s A)  (wtyped s A).
40  Proof. exact (@retract_has_ty s A). Defined.
41
42  Instance retract_has_ty_rev_instance s A:
43    ImpRev (wtyped (inj s) A) (wtyped_plus (inj s) A).
44  Proof. exact (@retract_has_ty_rev s A). Defined.
45
46  MetaCoq Run
47  Modular Lemma pres_plus where (exp_plus exp) extends exp at 0
48     with [~ eval_plus ~> eval ~] :
49    forall e e', eval_plus e e' -> forall A, wtyped e A -> wtyped e' A.
50  Next Obligation.
51    intros IH e e' Heval. destruct Heval; intros.
52    - minversion H1. mconstructor. eapply IH; eassumption. eassumption.
53    - minversion H1. mconstructor. eassumption. eapply IH; eassumption.
54    - minversion H0. mconstructor.
55  Defined.
56
57  End Plus.
58
59  (** ** Option expressions *)
```

MAIN 2 SHELVED 0 GIVEN UP 0

```
exp :  Type
H :  included exp_plus exp
wtyped :  exp -> ty -> Prop
eval :  exp -> exp -> Prop
retract_has_ty :  forall (s : exp) (A : ty),
                  wtyped_plus s A -> wtyped s A
retract_has_ty_rev :  forall (s : exp_plus exp) (A : ty),
                      wtyped (inj s) A ->
                      wtyped_plus (inj s) A

pres,
IH :  forall e e' : exp,
      eval e e' ->
      forall A : ty, wtyped e A -> wtyped e' A
e1, e1', e2 :  exp
H0 :  eval e1 e1'
H1 :  wtyped_plus (inj (plus exp e1 e2)) TNat
H3 :  wtyped e1 TNat
H4 :  wtyped e2 TNat

(1/2)
wtyped e1' TNat

(2/2)
wtyped e2 TNat
```

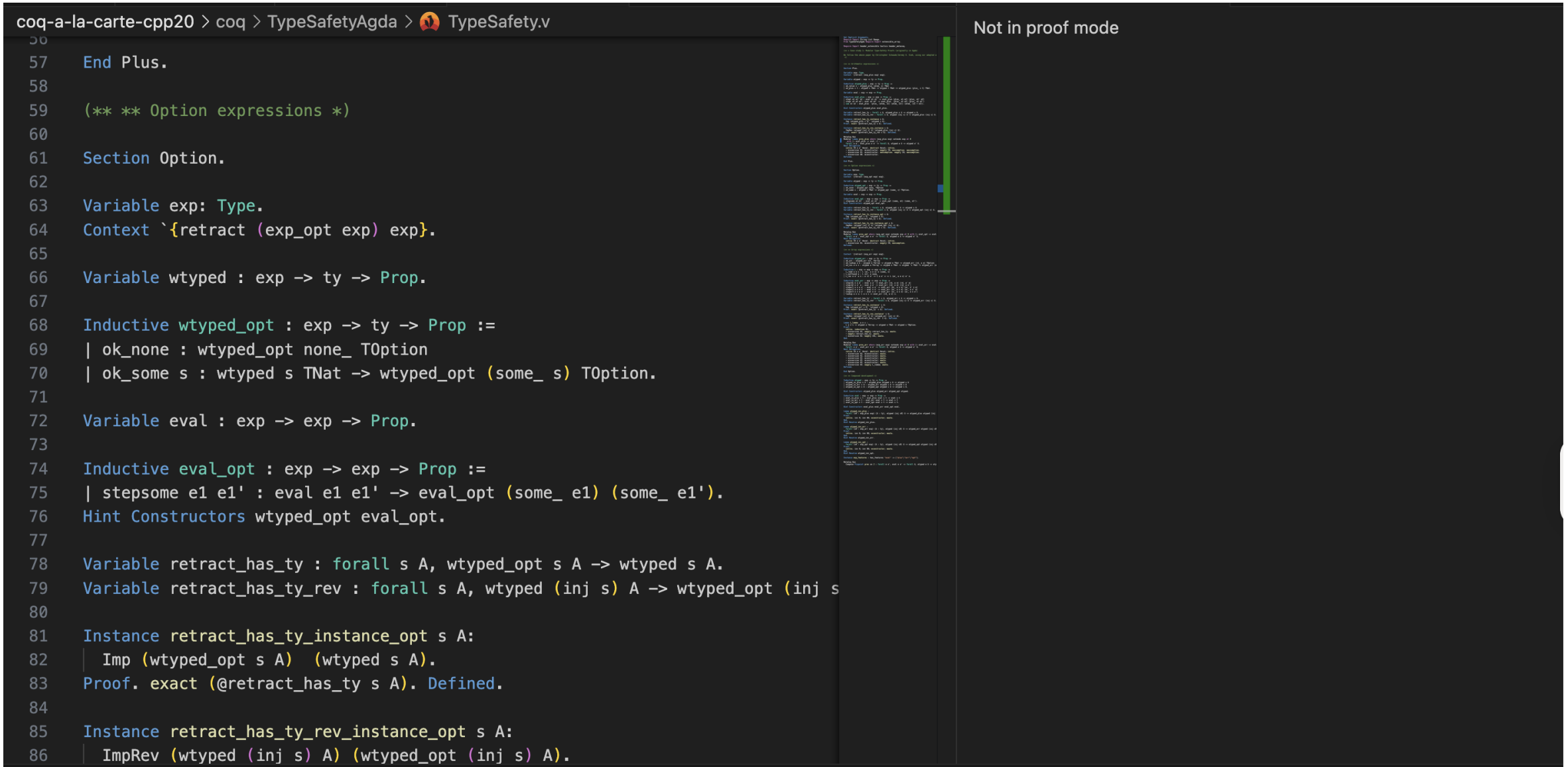
Demo

User-defined Parts: A Proof of Type Safety



Demo

User-defined Parts: A Proof of Type Safety



Demo

User-defined Parts: A Proof of Type Safety (Composition)


```
Lemma wtyped_inv_plus :
  forall (s0 : exp_plus exp) (A : ty), wtyped (inj s0) A
    -> wtyped_plus wtyped (inj s0) A.
Proof.
  intros. inv H; inv H0; econstructor; eauto.
Qed.
Hint Resolve wtyped_inv_plus.

Lemma wtyped_inv_arr :
  forall (s0 : exp_arr exp) (A : ty), wtyped (inj s0) A
    -> wtyped_arr wtyped (inj s0) A.
Proof.
  intros. inv H; inv H0; econstructor; eauto.
Qed.
Hint Resolve wtyped_inv_arr.

Lemma wtyped_inv_opt :
  forall (s0 : exp_opt exp) (A : ty), wtyped (inj s0) A
    -> wtyped_opt wtyped (inj s0) A.
Proof.
  intros. inv H; inv H0; econstructor; eauto.
Qed.
Hint Resolve wtyped_inv_opt.

Instance exp_features : has_features "eval" := ["plus";"arr";"opt"].

MetaCoq Run
  Compose Fixpoint pres on 2 : forall e e', eval e e'
    -> forall A, wtyped e A -> wtyped e' A.
```

How Far Does This Scale?

LOC-wise: A comparably small overhead compared to a non-modular development; achieved by a mix of library & metaprogramming

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].

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.

... and all of this can be combined with the substitution boilerplate generation of Autosubst.

But…

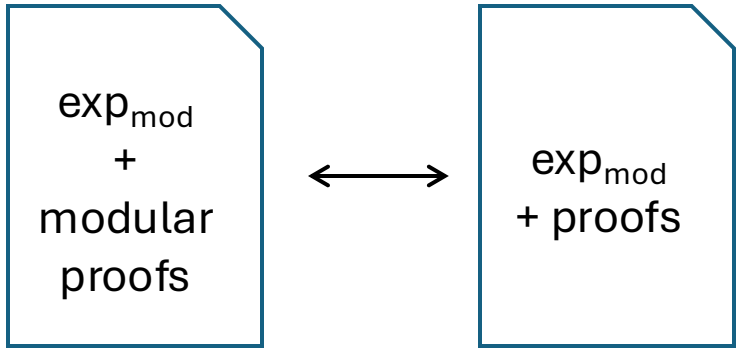
Dependency on specific version of MetaRocq – complicated installation; not working with current version of MetaRocq

Very much a proof of concept; more mature use of tactics

Example of strong normalization – but missing a bigger library; e.g. building on an existing development

There are still parts that resemble a modular proof – can we eliminate them?

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.



More Recent Related Work

How different is a proof allowed to be to
- develop a modular library
- use a modular library?

- Lean library using Lean's metaprogramming facilities [Shahin, Types '25]
- Based on Family Polymorphism:
  - Extensible Metatheory Mechanization via Family Polymorphism [Jin et al., PLDI '23]
  - Persimmon: Nested Family Polymorphism with Extensible Variant Types [Kravchuk-Kirilyuk, OOPSLA '24]
  - Certified Compilers a la Carte [Ebresafe et al., PLDI '25]

Today’s Talk

1. Autosubst – a library for binders
2. Towards a library for modular proofs

Thanks for your attention!

Questions?