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

Yannick Forster and Kathrin Stark

SAARLAND UNIVERSITY

SIC Saarland Informatics Campus

PRINCETON UNIVERSITY

CPP 2020, January 21

# Our Motivation: The Expression Problem [Wadler, 2003]

- 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 the terms.

- ... and then want to **extend** this calculus, e.g. by boolean expressions:

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

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

# Our Motivation: The Expression Problem[Wadler, 2003]

- 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 the terms.

- ... and then want to **extend** this calculus, e.g. by boolean expressions:

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

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

# Our Motivation: The Expression Problem[Wadler, 2003]

- 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 the terms.

- ...and then want to **extend** this calculus, e.g. by boolean expressions:

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

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

# Contribution: An Approach to Modular Syntax

- A solution to the expression problem in Coq
- Scales to proofs of **preservation** and **strong normalisation**

# Related Work (I)

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

- Features as functors, e.g.[1]

  $\texttt{Inductive exp}_\lambda\ (\texttt{exp} : \texttt{Type}) :=$
  $\mid\ \texttt{var} :\ \texttt{nat} \to \texttt{exp}_\lambda\ \texttt{exp}$
  $\mid\ \texttt{app} :\ \texttt{exp} \to \texttt{exp} \to \texttt{exp}_\lambda\ \texttt{exp}$
  $\mid\ \texttt{abs} :\ \texttt{exp} \to \texttt{exp}_\lambda\ \texttt{exp}.$

- A general expression type as fixed point of functors:

  $\texttt{Inductive exp}\ (\texttt{F} : \texttt{Type} \to \texttt{Type}) : \texttt{Type} :=$
  $\mid\ \texttt{In} :\ \texttt{F}\ (\texttt{exp F}) \to \texttt{exp F}.$

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

- Functions = algebras, assembling via type classes

[1] We use Coq syntax for convenience.

# Related Work (I)

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

- Features as functors, e.g.[1]

```
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

[1] We use Coq syntax for convenience.

# Related Work (I)

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

- Features as functors, e.g.[1]

```
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.
```

```
uU:---  *goals*        All L1      (Coq Goals Utoks)
Error: Non strictly positive occurrence of "exp" 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

[1] We use Coq syntax for convenience.

# 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 à la Carte [Delaware et al., 2013]
- Generic Data Types à la Carte [Keuchel et al., 2013]
- Modular Monadic Meta-Theory [Delaware et al., 2013]

# 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 à la Carte [Delaware et al., 2013]
- Generic Data Types à la Carte [Keuchel et al., 2013]
- Modular Monadic Meta-Theory [Delaware et al., 2013]

# Related Work (III)

## Why are there no developments based on these approaches?

These developments are **truly modular**, but not **practical**[Aydemir et al., 2005]:

- Many introductory statements and intermediate proofs — failing conciseness
- Encoded definitions — failing transparency
- Encoded definitions and intermediate tactics — failing accessibility

**Problem:** Encoding of the functor adds a layer of indirectness and Coq's tactic support fails.

# Related Work (III)

## Why are there no developments based on these approaches?

These developments are **truly modular**, but not **practical**[Aydemir et al., 2005]:

- Many introductory statements and intermediate proofs — failing conciseness
- Encoded definitions — failing transparency
- Encoded definitions and intermediate tactics — failing accessibility

**Problem:** Encoding of the functor adds a layer of indirectness and Coq's tactic support fails.

# Related Work (III)

## Why are there no developments based on these approaches?

These developments are **truly modular**, but not **practical**[Aydemir et al., 2005]:

- Many introductory statements and intermediate proofs — failing conciseness
- Encoded definitions — failing transparency
- Encoded definitions and intermediate tactics — failing accessibility

**Problem:** Encoding of the functor adds a layer of indirectness and Coq's tactic support fails.

# Contribution: A Practical Approach to Modular Syntax

- Modular syntax via variants with direct injections:

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

⇒

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

- **Tool support:**
  - Boilerplate generation with an extension of Autosubst 2
  - Assembling via MetaCoq[Sozeau et al., 2019]
- **Result:**
  - Practical modular developments
  - Improvement in case study of Delaware et al. from 1000 loc/feature to 125 loc/feature

# Contribution: A Practical Approach to Modular Syntax

- Modular syntax via variants with direct injections:

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

⇒

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

- **Tool support:**
  - Boilerplate generation with an extension of Autosubst 2
  - Assembling via MetaCoq[Sozeau et al., 2019]
- **Result:**
  - Practical modular developments
  - Improvement in case study of Delaware et al. from 1000 loc/feature to 125 loc/feature

# Contribution: A Practical Approach to Modular Syntax

- Modular syntax via variants with direct injections:

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

⇒

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

- **Tool support:**
  - Boilerplate generation with an extension of Autosubst 2
  - Assembling via MetaCoq[Sozeau et al., 2019]
- **Result:**
  - Practical modular developments
  - Improvement in case study of Delaware et al. from 1000 loc/feature to 125 loc/feature

# Modular Syntax

# Framework

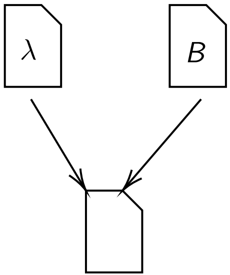
1. Mechanisation of features.
   - Parameterised by variants.
   - Unchanged for new variants.
2. Mechanisation of variants.

# Framework


1. Mechanisation of features.
   - Parameterised by variants.
   - Unchanged for new variants.
2. Mechanisation of variants.

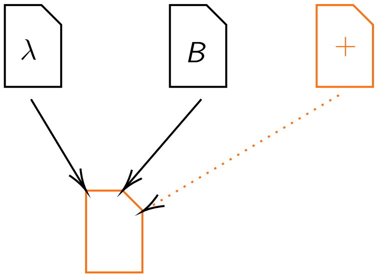
# Framework

1. Mechanisation of features.
   - Parameterised by variants.
   - Unchanged for new variants.
2. Mechanisation of variants.



# Framework

1. Mechanisation of features.
   - Parameterised by variants.
   - Unchanged for new variants.
2. Mechanisation of variants.



# Modular Expressions

1. Define features, parameterised by the variants.

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

2. Define variants.

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

3. What is the connection between features and variants?

# Modular Expressions

1. Define features, parameterised by the variants.

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

2. Define variants.

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

3. What is the connection between features and variants?

# Connection via a Retract

$\texttt{inj} : \texttt{exp}_\lambda\ \texttt{exp} \to \texttt{exp}$

$\texttt{exp}_\lambda\ \texttt{exp}$ $\qquad$ $\texttt{exp}$

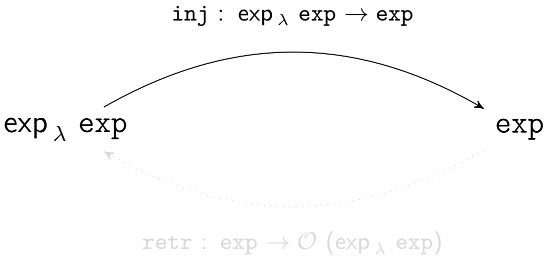
$\texttt{retr} : \texttt{exp} \to \mathcal{O}\ (\texttt{exp}_\lambda\ \texttt{exp})$

If retr y = Some x, then inj x = y.

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

$\texttt{app} : \texttt{exp} \to \texttt{exp} \to \texttt{exp}_\lambda\ \texttt{exp}$
$\texttt{app\_} : \texttt{exp} \to \texttt{exp} \to \texttt{exp}$
$\texttt{app\_ s t} := \texttt{inj (app s t)}$

# Connection via a Retract



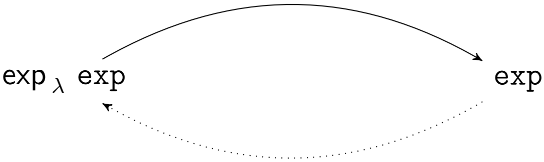

If retr y = Some x, then inj x = y.

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

app : exp → exp → exp λ exp
app_ : exp → exp → exp
app_ s t := inj (app s t)

# Connection via a Retract

$\texttt{inj} : \texttt{exp} \;\lambda\; \texttt{exp} \to \texttt{exp}$




$\texttt{retr} : \texttt{exp} \to \mathcal{O}\ (\texttt{exp} \;\lambda\; \texttt{exp})$

If retr y = Some x, then inj x = y.

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

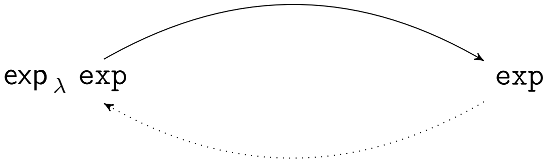
$\texttt{app} : \texttt{exp} \to \texttt{exp} \to \texttt{exp} \;\lambda\; \texttt{exp}$

$\texttt{app\_} : \texttt{exp} \to \texttt{exp} \to \texttt{exp}$

$\texttt{app\_ s t} := \texttt{inj (app s t)}$

# Connection via a Retract

inj : exp $_\lambda$ exp → exp



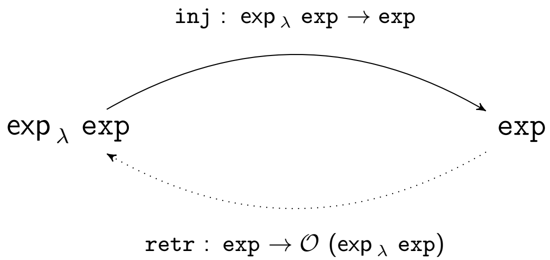

retr : exp → $\mathcal{O}$ (exp $_\lambda$ exp)

If retr y = Some x, then inj x = y.

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

app : exp → exp → exp $_\lambda$ exp
app_ : exp → exp → exp
app_ s t := inj (app s t)

# Connection via a Retract

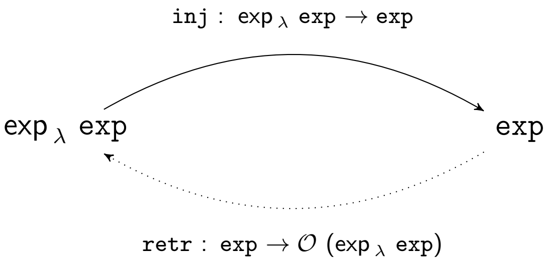



If retr y = Some x, then inj x = y.

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

app : exp → exp → exp $_\lambda$ exp
app_ : exp → exp → exp
app_ s t := inj (app s t)

# Connection via a Retract




If retr y = Some x, then inj x = y.

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

$$\texttt{app} : \texttt{exp} \to \texttt{exp} \to \texttt{exp}_\lambda\ \texttt{exp}$$
$$\texttt{app\_} : \texttt{exp} \to \texttt{exp} \to \texttt{exp}$$
$$\texttt{app\_ s t} := \texttt{inj (app s t)}$$

# Modular Recursive Functions

1. Define feature functions, parameterised by the variant functions.
2. Define variant functions.
3. What is the connection between feature function and variant function?

# Modular Recursive Functions

1. Define feature functions, parameterised by the variant functions.

```
Variable |_| : exp → nat.
Definition |_|_λ : exp_λ exp → nat :=
  fun e ⇒ match e with
  | var x ⇒ 1
  | λ.s ⇒ |s|
  | app s t ⇒ |s| + |t|
  end.
```

2. Define variant functions.
3. What is the connection between feature function and variant function?

# Modular Recursive Functions

1. Define feature functions, parameterised by the variant functions.

```
Variable |_| : exp → nat.
Definition |_|λ : exp λ exp → nat :=
  fun e ⇒ match e with
  | var x ⇒ 1
  | λ.s ⇒ |s|
  | app s t ⇒ |s| + |t|
  end.
```

2. Define variant functions.

```
Fixpoint |_| (e : exp) : nat :=
match e with
| injλ e ⇒ |e|λ^|_|
| inj𝔹 e ⇒ |e|𝔹^|_|
end.
```

3. What is the connection between feature function and variant function?

# Modular Recursive Functions

1. Define feature functions, parameterised by the variant functions.

```
Variable |_| : exp → nat.
Definition |_|_λ : exp_λ exp → nat :=
  fun e ⇒ match e with
  | var x ⇒ 1
  | λ.s ⇒ |s|
  | app s t ⇒ |s| + |t|
  end.
```

2. Define variant functions.

```
Fixpoint |_| (e : exp) : nat :=
match e with
| inj_λ e ⇒ |e|_λ^{|_|}
| inj_𝔹 e ⇒ |e|_𝔹^{|_|}
end.
```

3. What is the connection between feature function and variant function?

$$\forall (s : \mathrm{exp}_i\ \mathrm{exp}).|s|_i = |\mathrm{inj}_i\ s|$$

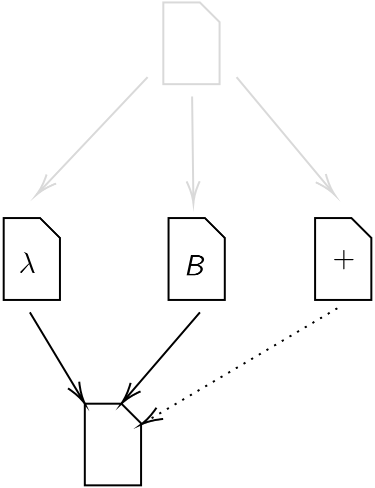
# Modular Inductive Proofs

1. Define feature proofs.
2. Define variant proofs.
3. Connection between feature proof and variant proof?

# Modular Inductive Predicates over Modular Syntax

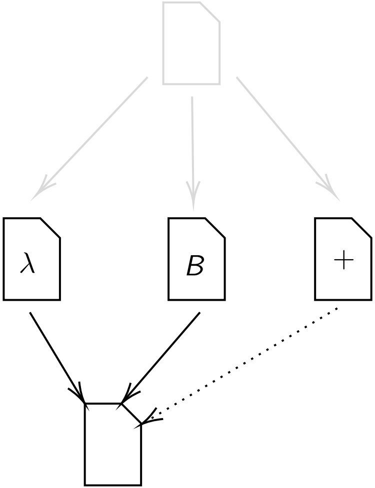
1. Define feature predicate.
2. Define variant predicate.
3. Connection between feature inductive predicate and variant inductive predicate?

# General Framework and Tool Support



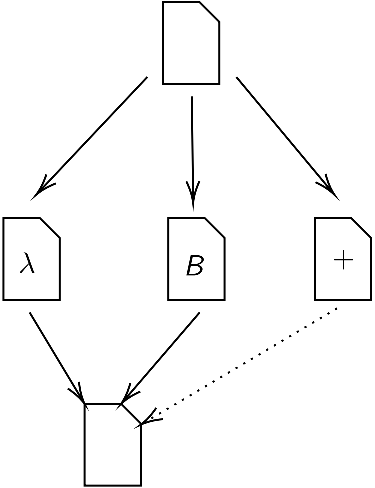

- **The good:**
  - Transparent and accessible
  - Implementable in any proof assistant
  - Truly modular*
    - ★ Termination has to be rechecked, custom induction principles change this
- **The bad:**
  1. Preliminary definitions for retracts, smart constructors, and induction principles
  2. Combination of definitions
  3. Proofs might require additional steps

# General Framework and Tool Support




- **The good:**
  - Transparent and accessible
  - Implementable in any proof assistant
  - Truly modular*
    - Termination has to be rechecked, custom induction principles change this
- **The bad:**
  1. Preliminary definitions for retracts, smart constructors, and induction principles
  2. Combination of definitions
  3. Proofs might require additional steps

# General Framework and Tool Support

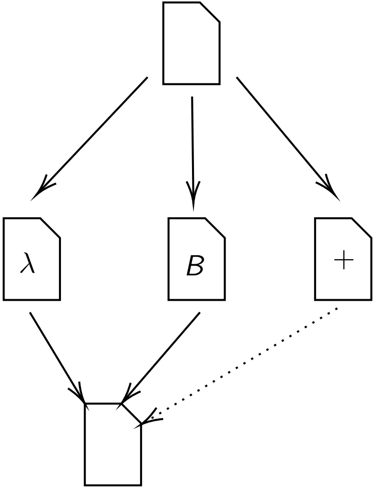



1.) Header file,
generated by Autosubst
according to a specification

3.) Simple tactic support for
simplification, constructor,
and inversion

2.) MetaCoq support for
combining functions/proofs

# General Framework and Tool Support

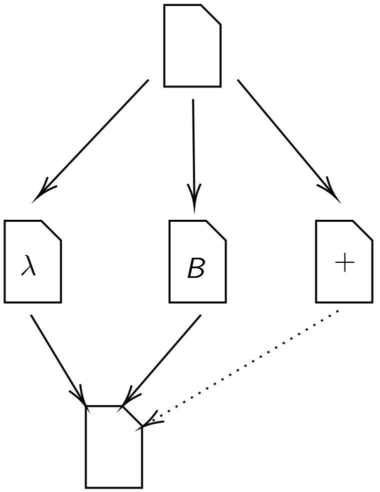



1.) Header file,
generated by Autosubst
according to a specification

3.) Simple tactic support for
simplification, constructor,
and inversion

2.) MetaCoq support for
combining functions/proofs

# General Framework and Tool Support




1.) Header file,
generated by Autosubst
according to a specification

3.) Simple tactic support for
simplification, constructor,
and inversion

2.) MetaCoq support for
combining functions/proofs

# Demo

# Start with a specification...

```
begin lam
  arr : ty → ty → ty
  ab : ty → (exp → exp) → exp
  app : exp → exp → exp
end lam

begin bool
  boolTy : ty
  constBool : bool → exp
  If : exp → exp → exp → exp
end bool
```

```
begin arith
  natTy : ty
  plus : exp → exp → exp
  constNat : nat → exp
end arith

compose lambdas := lam
compose all := lam :+: bool :+: arith
```

...import the generated file in the feature file and prove preservation for the lambda feature...

File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outlin

State Context Goal Retract Undo Next U

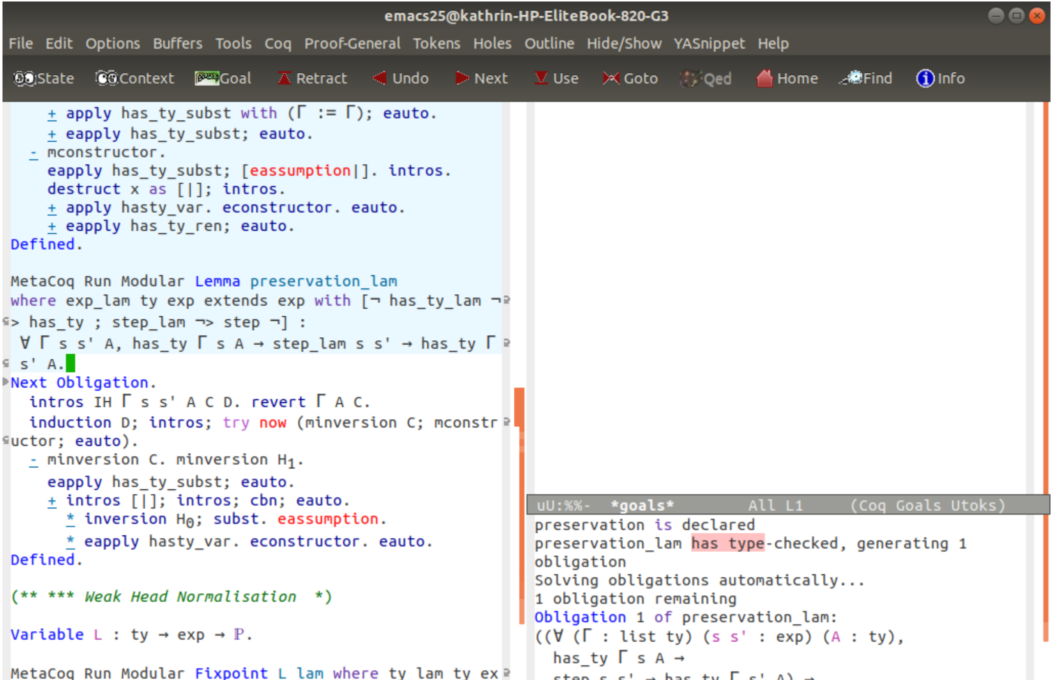
```
Require Export tactics header_metacoq.
From SN Require Export expressions defs.
Require Export FunctionalExtensionality.


(** ** mini-ML: Λ expressions *)

Section MiniML_Λ.

Variable ty : Type.
Context {HT_lam : included ty_lam ty}.

Variable exp : Type.
Context `{Hr : retract exp_var exp}.
Context `{retract (exp_lam ty exp) exp}.

Hint Rewrite retract_works : retract_forward.
```

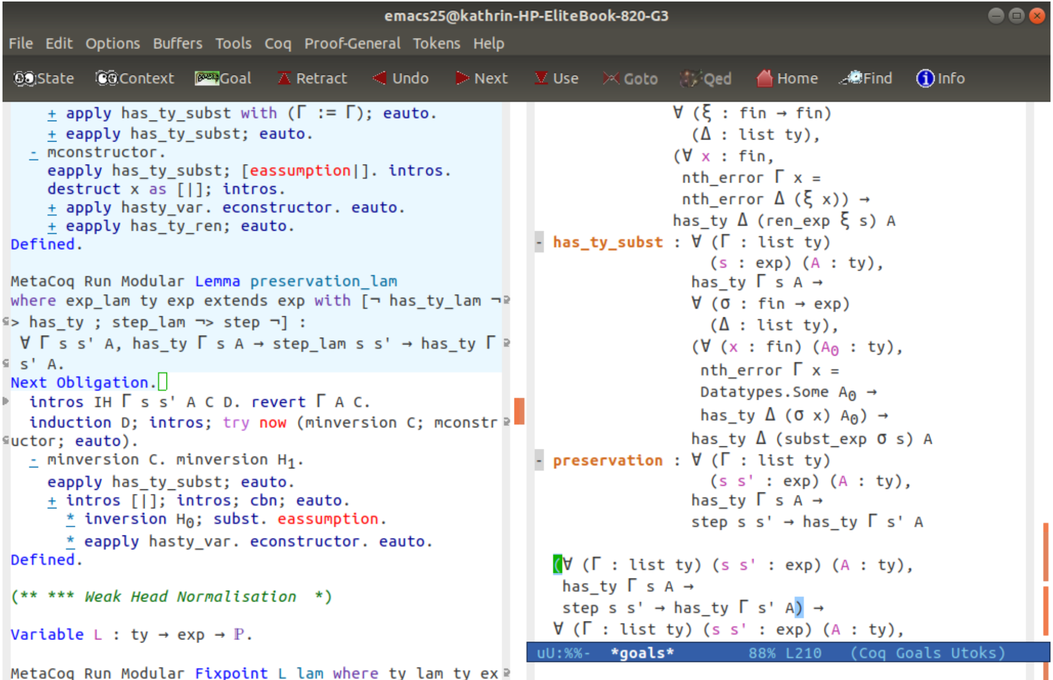
emacs25@kathrin-HP-EliteBook-820-G3

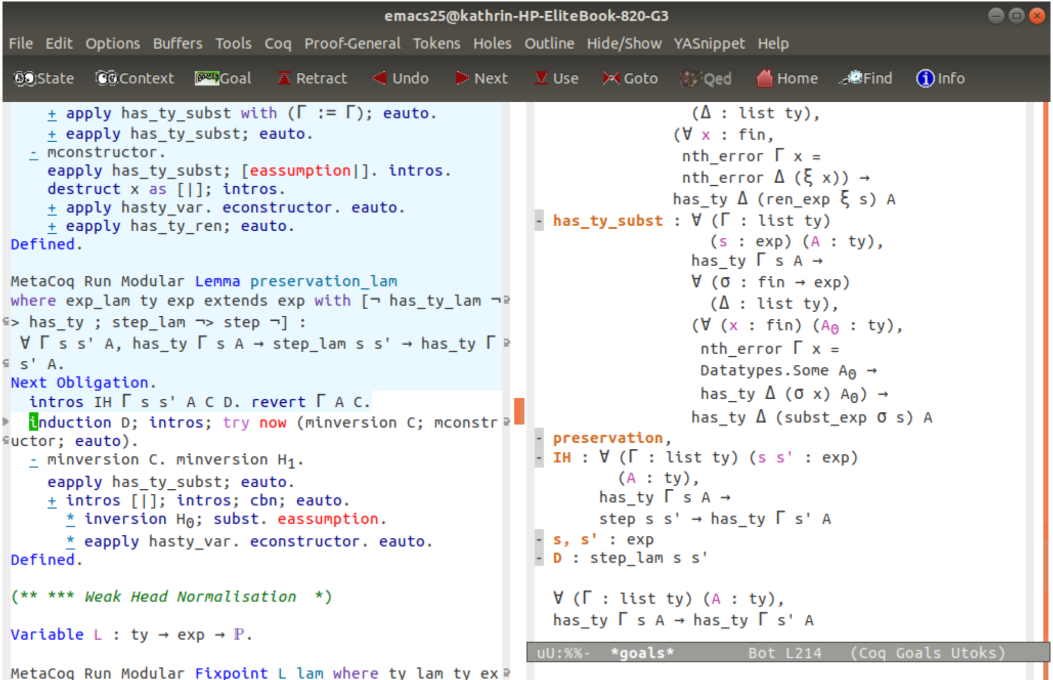
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help

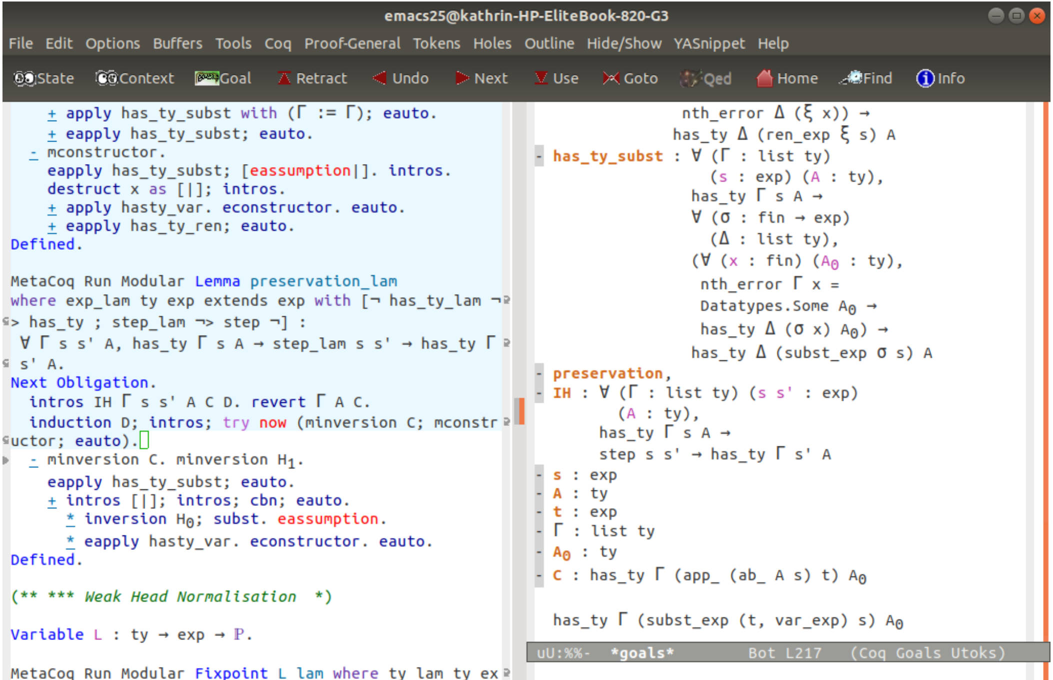
State Context Goal Retract Undo Next Use Goto Qed Home Find Info

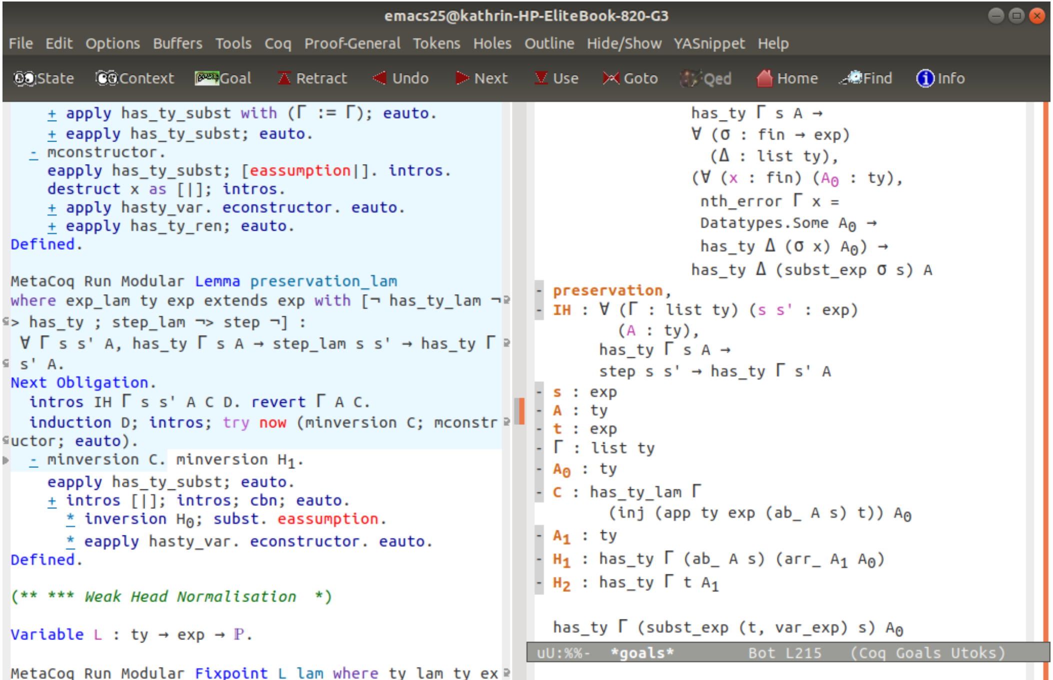
```
Hint Rewrite retract_ren_exp retract_subst_exp  up_exp_exp_def up_exp_exp_def': retract_forward.
Hint Rewrite retract_ren_exp retract_subst_exp up_exp_exp_def up_exp_exp_def': retract_rev.

(** *** Preservation *)
Variable has_ty : list ty → exp → ty → ℙ.

Inductive has_ty_lam (Γ : list ty) : exp → ty → ℙ :=
| hasty_app (A B: ty) s t : has_ty Γ s (arr_ A B) → has_ty Γ t A → has_ty_lam Γ (app_ s t) B
| hasty_lam A B s: has_ty (A :: Γ) s B →  has_ty_lam Γ (ab_ A s) (arr_ A B).

Require Import SN.sn_var.

Variable  hasty_var : ∀ Γ s A, has_ty_var _ _ Γ s A → has_ty Γ s A.

Variable retract_has_ty : ∀ Γ s A, has_ty_lam Γ s A → has_ty Γ s A.
Variable retract_has_ty_rev : ∀ Γ s A, has_ty Γ (inj s) A → has_ty_lam Γ (inj s) A.

Instance retract_has_ty_instance Γ s A:
  Imp (has_ty_lam Γ s A) (has_ty Γ s A).
Proof. exact (retract_has_ty Γ s A). Defined.

Instance retract_has_ty_rev_instance Γ s A:
  ImpRev (has_ty Γ (inj s) A) (has_ty_lam Γ (inj s) A).
Proof. exact (retract_has_ty_rev Γ s A). Defined.

Variable step: exp → exp → ℙ.

Inductive step_lam : exp → exp → ℙ :=
| stepBeta (s: exp) A t: step_lam (app_ (ab_ A s) t) (subst_exp (scons t var_exp) s)
```

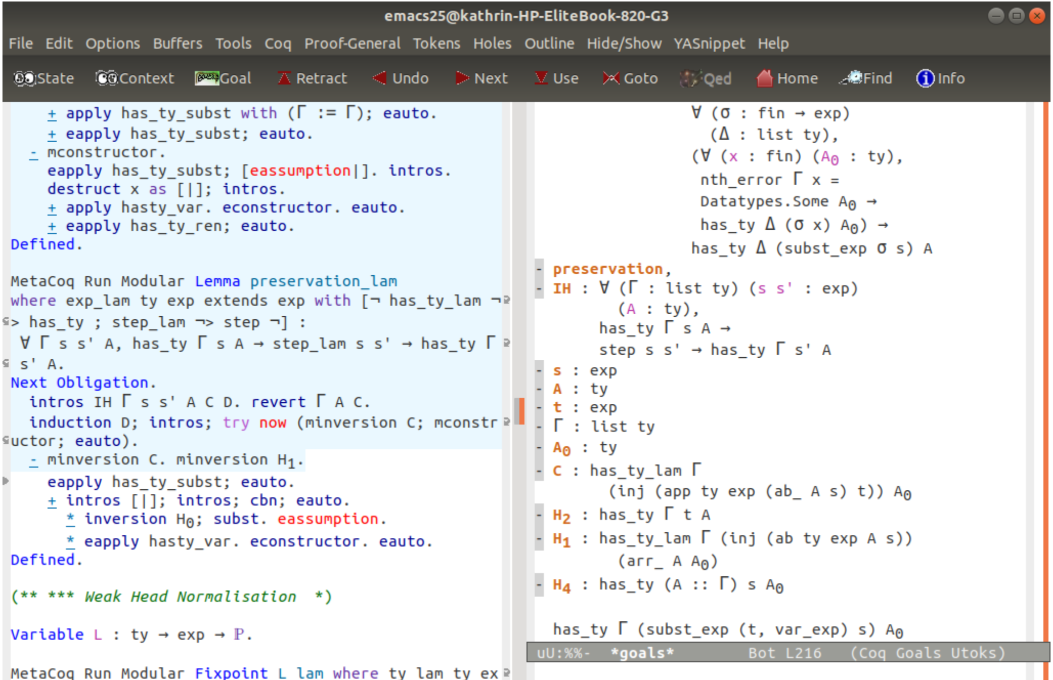

emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬
> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ
s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstr
uctor; eauto).
- minversion C. minversion H1.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion H0; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L lam where ty lam ty ex
uU:%%- *goals* All L1 (Coq Goals Utoks)
preservation is declared
preservation_lam has type-checked, generating 1
obligation
Solving obligations automatically...
1 obligation remaining
Obligation 1 of preservation_lam:
((∀ (Γ : list ty) (s s' : exp) (A : ty),
has_ty Γ s A →

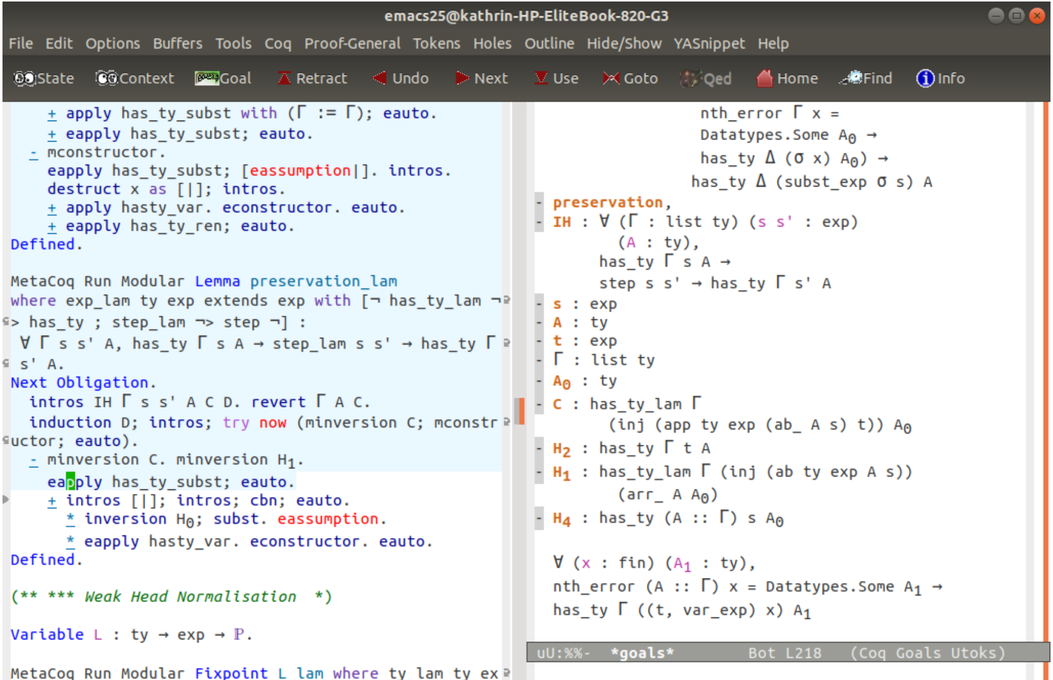

emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstructor; eauto).
- minversion C. minversion H1.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion H0; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L lam where ty lam ty ex
∀ (ξ : fin → fin)
(Δ : list ty),
(∀ x : fin,
nth_error Γ x =
nth_error Δ (ξ x)) →
has_ty Δ (ren_exp ξ s) A
- has_ty_subst : ∀ (Γ : list ty)
(s : exp) (A : ty),
has_ty Γ s A →
∀ (σ : fin → exp)
(Δ : list ty),
(∀ (x : fin) (A0 : ty),
nth_error Γ x =
Datatypes.Some A0 →
has_ty Δ (σ x) A0) →
has_ty Δ (subst_exp σ s) A
- preservation : ∀ (Γ : list ty)
(s s' : exp) (A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A
(∀ (Γ : list ty) (s s' : exp) (A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A) →
∀ (Γ : list ty) (s s' : exp) (A : ty),
uU:%%- *goals* 88% L210 (Coq Goals Utoks)

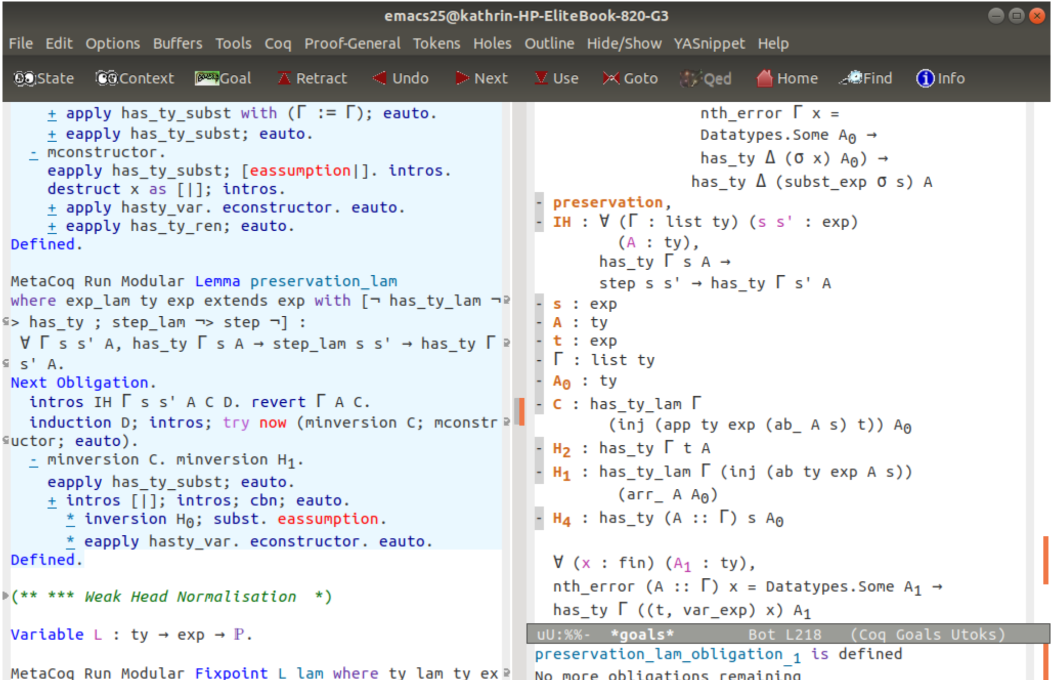

emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstructor; eauto).
- minversion C. minversion H1.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion H0; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L lam where ty lam ty ex
(Δ : list ty),
(∀ x : fin,
nth_error Γ x =
nth_error Δ (ξ x)) →
has_ty Δ (ren_exp ξ s) A
has_ty_subst : ∀ (Γ : list ty)
(s : exp) (A : ty),
has_ty Γ s A →
∀ (σ : fin → exp)
(Δ : list ty),
(∀ (x : fin) (A0 : ty),
nth_error Γ x =
Datatypes.Some A0 →
has_ty Δ (σ x) A0) →
has_ty Δ (subst_exp σ s) A
preservation,
IH : ∀ (Γ : list ty) (s s' : exp)
(A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A
s, s' : exp
D : step_lam s s'
∀ (Γ : list ty) (A : ty),
has_ty Γ s A → has_ty Γ s' A
uU:%%- *goals* Bot L214 (Coq Goals Utoks)


emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬
> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ
s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstr
uctor; eauto).
- minversion C. minversion H1.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion H0; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L lam where ty lam ty ex
nth_error Δ (ξ x)) →
has_ty Δ (ren_exp ξ s) A
has_ty_subst : ∀ (Γ : list ty)
(s : exp) (A : ty),
has_ty Γ s A →
∀ (σ : fin → exp)
(Δ : list ty),
(∀ (x : fin) (A0 : ty),
nth_error Γ x =
Datatypes.Some A0 →
has_ty Δ (σ x) A0) →
has_ty Δ (subst_exp σ s) A
preservation,
IH : ∀ (Γ : list ty) (s s' : exp)
(A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A
s : exp
A : ty
t : exp
Γ : list ty
A0 : ty
C : has_ty Γ (app_ (ab_ A s) t) A0
has_ty Γ (subst_exp (t, var_exp) s) A0
uU:%%- *goals* Bot L217 (Coq Goals Utoks)


emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstructor; eauto).
- minversion C. minversion $H_1$.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion $H_0$; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L lam where ty lam ty ex
has_ty Γ s A →
∀ (σ : fin → exp)
(Δ : list ty),
(∀ (x : fin) ($A_0$ : ty),
nth_error Γ x =
Datatypes.Some $A_0$ →
has_ty Δ (σ x) $A_0$) →
has_ty Δ (subst_exp σ s) A
preservation,
IH : ∀ (Γ : list ty) (s s' : exp)
(A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A
s : exp
A : ty
t : exp
Γ : list ty
$A_0$ : ty
C : has_ty_lam Γ
(inj (app ty exp (ab_ A s) t)) $A_0$
$A_1$ : ty
$H_1$ : has_ty Γ (ab_ A s) (arr_ $A_1$ $A_0$)
$H_2$ : has_ty Γ t $A_1$
has_ty Γ (subst_exp (t, var_exp) s) $A_0$
uU:%%- *goals* Bot L215 (Coq Goals Utoks)


emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬
> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ
s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstr
uctor; eauto).
- minversion C. minversion H1.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion H0; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L lam where ty lam ty ex
∀ (σ : fin → exp)
(Δ : list ty),
(∀ (x : fin) (A0 : ty),
nth_error Γ x =
Datatypes.Some A0 →
has_ty Δ (σ x) A0) →
has_ty Δ (subst_exp σ s) A
preservation,
IH : ∀ (Γ : list ty) (s s' : exp)
(A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A
s : exp
A : ty
t : exp
Γ : list ty
A0 : ty
C : has_ty_lam Γ
(inj (app ty exp (ab_ A s) t)) A0
H2 : has_ty Γ t A
H1 : has_ty_lam Γ (inj (ab ty exp A s))
(arr_ A A0)
H4 : has_ty (A :: Γ) s A0
has_ty Γ (subst_exp (t, var_exp) s) A0
uU:%%- *goals* Bot L216 (Coq Goals Utoks)


emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstructor; eauto).
- minversion C. minversion H1.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion H0; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L_lam where ty_lam ty ex
nth_error Γ x =
Datatypes.Some A0 →
has_ty Δ (σ x) A0) →
has_ty Δ (subst_exp σ s) A
preservation,
IH : ∀ (Γ : list ty) (s s' : exp)
(A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A
s : exp
A : ty
t : exp
Γ : list ty
A0 : ty
C : has_ty_lam Γ
(inj (app ty exp (ab_ A s) t)) A0
H2 : has_ty Γ t A
H1 : has_ty_lam Γ (inj (ab ty exp A s))
(arr_ A A0)
H4 : has_ty (A :: Γ) s A0
∀ (x : fin) (A1 : ty),
nth_error (A :: Γ) x = Datatypes.Some A1 →
has_ty Γ ((t, var_exp) x) A1
uU:%%- *goals* Bot L218 (Coq Goals Utoks)


emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info
+ apply has_ty_subst with (Γ := Γ); eauto.
+ eapply has_ty_subst; eauto.
- mconstructor.
eapply has_ty_subst; [eassumption|]. intros.
destruct x as [|]; intros.
+ apply hasty_var. econstructor. eauto.
+ eapply has_ty_ren; eauto.
Defined.
MetaCoq Run Modular Lemma preservation_lam
where exp_lam ty exp extends exp with [¬ has_ty_lam ¬> has_ty ; step_lam ¬> step ¬] :
∀ Γ s s' A, has_ty Γ s A → step_lam s s' → has_ty Γ s' A.
Next Obligation.
intros IH Γ s s' A C D. revert Γ A C.
induction D; intros; try now (minversion C; mconstructor; eauto).
- minversion C. minversion H1.
eapply has_ty_subst; eauto.
+ intros [|]; intros; cbn; eauto.
* inversion H0; subst. eassumption.
* eapply hasty_var. econstructor. eauto.
Defined.
(** *** Weak Head Normalisation *)
Variable L : ty → exp → ℙ.
MetaCoq Run Modular Fixpoint L lam where ty lam ty ex
nth_error Γ x =
Datatypes.Some A0 →
has_ty Δ (σ x) A0) →
has_ty Δ (subst_exp σ s) A
- preservation,
- IH : ∀ (Γ : list ty) (s s' : exp)
(A : ty),
has_ty Γ s A →
step s s' → has_ty Γ s' A
- s : exp
- A : ty
- t : exp
- Γ : list ty
- A0 : ty
- C : has_ty_lam Γ
(inj (app ty exp (ab_ A s) t)) A0
- H2 : has_ty Γ t A
- H1 : has_ty_lam Γ (inj (ab ty exp A s))
(arr_ A A0)
- H4 : has_ty (A :: Γ) s A0
∀ (x : fin) (A1 : ty),
nth_error (A :: Γ) x = Datatypes.Some A1 →
has_ty Γ ((t, var_exp) x) A1
uU:%%- *goals* Bot L218 (Coq Goals Utoks)
preservation_lam_obligation_1 is defined
No more obligations remaining

...and then combine everything.

emacs25@kathrin-HP-EliteBook-820-G3

File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help

State Context Goal Retract Undo Next Use Goto Qed Home Find Info

```
From SN Require Export expressions sn_B sn_lam sn_var sn_arith.
Require Import header_metacoq.

(** ** Composition *)

(** *** Definition of typing *)

Inductive has_ty : list ty → exp → ty → ℙ :=
| inl_has_ty_var Γ s A:  has_ty_var _ _ Γ s A → has_ty Γ s A
| inl_has_ty_lam Γ s A : has_ty_lam _ _ has_ty Γ s A → has_ty Γ s A
| inl_has_ty_B Γ s A : has_ty_B _ _ has_ty Γ s A → has_ty Γ s A
| inl_has_ty_arith Γ s A : has_ty_arith _ _ has_ty Γ s A → has_ty Γ s A.
Instance has_ty_features : has_features "has_ty" := ["var";"lam";"B";"arith"].

Hint Constructors has_ty.
Hint Constructors has_ty_var.

Lemma has_ty_rev_var: ∀ (Gamma0 : list ty) (s0 : exp_var (* ty *) (* exp *)) (A0 : ty),
    has_ty Gamma0 (inj s0) A0 → has_ty_var _ _ Gamma0 (inj s0) A0.
Proof.
  intros. inversion H; subst; eauto; inversion H0.
Qed.

Lemma has_ty_rev_lam: ∀ (Gamma0 : list ty) (s0 : exp_lam ty exp) (A0 : ty),
    has_ty Gamma0 (inj s0) A0 → has_ty_lam _ _ has_ty Gamma0 (inj s0) A0.
Proof.
```

emacs25@kathrin-HP-EliteBook-820-G3

File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help

State Context Goal Retract Undo Next Use Goto Qed Home Find Info

```
From SN Require Export expressions sn_B sn_lam sn_var
 sn_arith.
Require Import header_metacoq.

(** ** Composition *)

(** *** Definition of typing *)

Inductive has_ty : list ty → exp → ty → ℙ :=
| inl_has_ty_var Γ s A:  has_ty_var _ _ Γ s A → has_
ty Γ s A
| inl_has_ty_lam Γ s A : has_ty_lam _ _ has_ty Γ s A
→ has_ty Γ s A
| inl_has_ty_B Γ s A : has_ty_B _ _ has_ty Γ s A → h
as_ty Γ s A
| inl_has_ty_arith Γ s A : has_ty_arith _ _ has_ty Γ
s A → has_ty Γ s A.
Instance has_ty_features : has_features "has_ty" := [
"var";"lam";"B";"arith"].

Hint Constructors has_ty.
Hint Constructors has_ty_var.

Lemma has_ty_rev_var: ∀ (Gamma0 : list ty) (s0 : exp_
var (* ty *) (* exp *)) (A0 : ty),
```

uU:--- *goals* All L1 (Coq Goals Utoks)

has_ty_features is defined

emacs25@kathrin-HP-EliteBook-820-G3

File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help

State Context Goal Retract Undo Next Use Goto Qed Home Find Info

```
(** *** Type preservation *)
MetaCoq Run Compose
Lemma preservation on 5 : ∀ Γ s s' A,
    has_ty Γ s A → step s s' → has_ty Γ s' A.
Hint Resolve preservation.

(** *** Weak Normalisation *)

Instance exp_features : has_features "exp" := ["var";"lam";"B";"arith"].
Instance ty_features : has_features "ty" := ["lam";"B";"arith"].

Fixpoint L (A : ty) : exp → ℙ :=
  match A with
  | In_ty_lam A ⇒ λ s ⇒ L_lam _ _ subst_exp step L A s
  | In_ty_B A ⇒ λ s ⇒ L_B  _ A s
  | In_ty_arith A ⇒ λ s ⇒ L_arith  _ A s
  end.

Definition E := E_ _ _ step L.

Fixpoint L_ren (s : exp) A ξ {struct A} :
  L A s → L A (ren_exp ξ s).
```

```
uU:---  *goals*        All L1       (Coq Goals Utoks)
has_ty_features is defined
```

emacs25@kathrin-HP-EliteBook-820-G3

File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help

State Context Goal Retract Undo Next Use Goto Qed Home Find Info

```
(** *** Type preservation *)
MetaCoq Run Compose
Lemma preservation on 5 : ∀ Γ s s' A,
    has_ty Γ s A → step s s' → has_ty Γ s' A.
Hint Resolve preservation.

(** *** Weak Normalisation *)

Instance exp_features : has_features "exp" := ["var";"lam";"B";"arith"].
Instance ty_features : has_features "ty" := ["lam";"B";"arith"].

Fixpoint L (A : ty) : exp → ℙ :=
  match A with
  | In_ty_lam A ⇒ λ s ⇒ L_lam _ _ subst_exp step L A s
  | In_ty_B A ⇒ λ s ⇒ L_B  _ A s
  | In_ty_arith A ⇒ λ s ⇒ L_arith  _ A s
  end.

Definition E := E_ _ _ step L.

Fixpoint L_ren (s : exp) A ξ {struct A} :
  L A s → L A (ren_exp ξ s).
```

# Case Study: Modular Proofs on the Meta-Theory of a Lambda Calculus

Proof of

- preservation;
- weak head normalisation
- and strong normalisation

for a $\lambda$-calculus with

- boolean expressions
- and arithmetic expressions

with 1000 loc for all features + 190 loc for the variant.

| 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 | - |

# Case Study: Modular Proofs on the Meta-Theory of a Lambda Calculus

Proof of

- preservation;
- weak head normalisation
- and strong normalisation

for a $\lambda$-calculus with

- boolean expressions
- and arithmetic expressions

with 1000 loc for all features + 190 loc for the variant.

| 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 | - |

# Wrap-Up

# Wrap-up

**Main Novel Results**

- A practical approach to truly modular syntax via feature functors and direct injections:
  - Support of simple inductive types, recursive functions, and inductive predicates
  - Usable both with and without tool support
  - Could cut previous case studies from 1000 loc/feature to 125 loc/feature

**Future Work**

- Explore new dimensions of modularity
- Tool support for scoped syntax and dependent predicates
  - Modular solutions to the POPLMark/POPLMark Reloaded challenge

**Available online:**

www.github.com/uds-psl/coq-a-la-carte-cpp20

# Wrap-up

## Main Novel Results

- A practical approach to truly modular syntax via feature functors and direct injections:
  - Support of simple inductive types, recursive functions, and inductive predicates
  - Usable both with and without tool support
  - Could cut previous case studies from 1000 loc/feature to 125 loc/feature

## Future Work

- Explore new dimensions of modularity
- Tool support for scoped syntax and dependent predicates
  - Modular solutions to the POPLMark/POPLMark Reloaded challenge

**Available online:**

www.github.com/uds-psl/coq-a-la-carte-cpp20

# Wrap-up

**Main Novel Results**

- A practical approach to truly modular syntax via feature functors and direct injections:
  - Support of simple inductive types, recursive functions, and inductive predicates
  - Usable both with and without tool support
  - Could cut previous case studies from 1000 loc/feature to 125 loc/feature

**Future Work**

- Explore new dimensions of modularity
- Tool support for scoped syntax and dependent predicates
  - Modular solutions to the POPLMark/POPLMark Reloaded challenge

**Available online:**

www.github.com/uds-psl/coq-a-la-carte-cpp20