Generating Infrastructural Code for Terms with Binders using MetaCoq

Bachelor Talk 1

Author: Adrian Dapprich
Advisor: Andrej Dudenhefner

Department of Computer Science
Saarland University

06. May 2021

Motivation

- Prove metatheorems (e.g. preservation, normalization) of programming languages modelled in Coq
- Reason about binders and substitution

  $(\lambda x.t)v \succ_\beta t[x \mapsto v]$

Problem

How do we model binders? (i.e. form of the λ constructor)

Solution

Autosubst (dissertation of Kathrin Stark [Stark, 2019])

Different Approaches

Named	[Barendregt, 1984]
Locally Nameless	[McBride and McKinna, 2004]
Anti-Locally Nameless	[Laurent, 2021]
De Bruijn Indices	[De Bruijn, 1972]
Co De Bruijn Indices	[McBride, 2018]
HOAS	[Pfenning and Elliott, 1988]
⋮	⋮

Binders & Substitutions

Based on Sigma Calculus [Abadi et al., 1991]

- De Bruijn indices

$$\lambda x.\lambda y.x \cong \lambda.\lambda.1$$

- Parallel substitutions replace all variables

$$(\lambda.0\ 1)\ 3 \succ_\beta (0\ 1)[3 \cdot id] \succ_\sigma 3\ 0$$

$$[3 \cdot id] \cong [0 \mapsto 3; 1 \mapsto 0; \ldots; n+1 \mapsto n; \ldots]$$

- Confluent and terminating rewriting system

$\to$ Can decide whether $s[\sigma] = t[\tau]$ ([Schäfer et al., 2015])

Substitution Lemmas

Prove any assumption-free substitution lemma[1]

Metatheorem

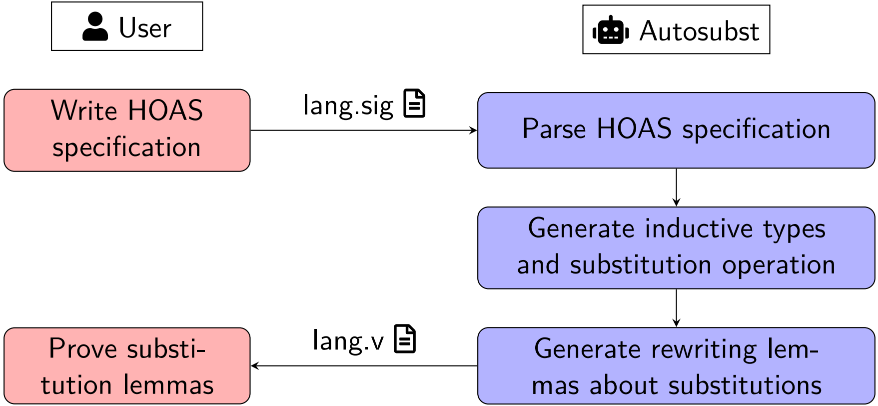
```
Lemma step_inst (σ: ℕ → tm) (s t: tm) :
  s ≻ t → s[σ] ≻ t[σ].
```

Goal in Proof

$$s[t \cdot id][\sigma] = s[\Uparrow \sigma][(t[\sigma]) \cdot id]$$
$$\Rightarrow s[t[\sigma] \cdot \sigma] = s[t[\sigma] \cdot \sigma]$$

[1] using functional extensionality

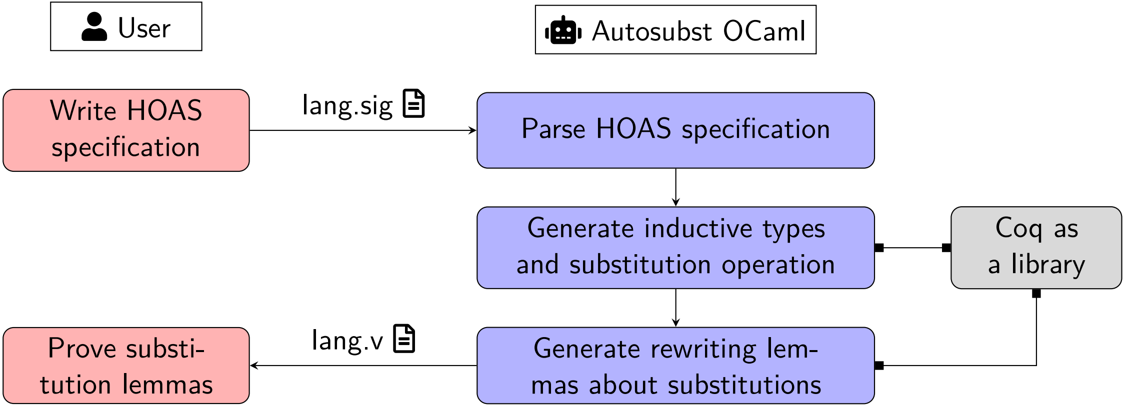
Workflow: Original Autosubst



Autosubst OCaml

- Reimplementation to use Coq as a library
- Use Coq AST and pretty printer to generate parseable code
- Tighter integration with Coq ecosystem

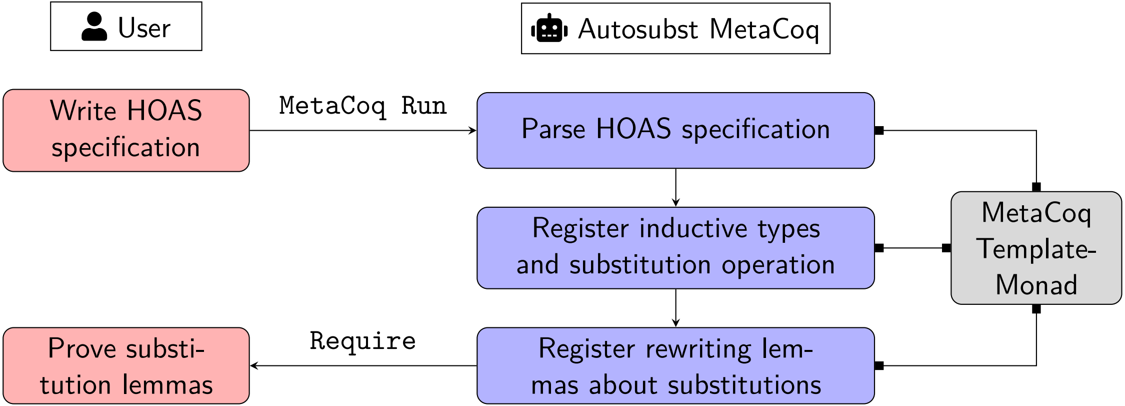
Workflow: Autosubst OCaml



Autosubst MetaCoq

- Reimplementation in MetaCoq
- Seamless usage from within Coq
- Even tighter integration with Coq ecosystem
- Correctness proofs might be possible

Workflow: Autosubst MetaCoq



HOAS [Pfenning and Elliott, 1988]

Used purely as notation for specifying syntax

Simply Typed Lambda Calculus

```
ty : Type
tm : Type

base : ty
arr  : ty → ty → ty

app : tm → tm → tm
lam : ty → (tm → tm) → tm
```

HOAS Parsing 1

(Custom Entry) Notations

```
Definition stlc_ctors :=
    {{ base : ty
       arr : ty → ty → ty;
       app : tm → tm → tm;
       lam : ty → (tm → tm) → tm }}.
```

- Deep embedding of HOAS
- Easy to use
- String handling behind the scenes is “unpalatable” ([Pit-Claudel and Bourgeat,])

HOAS Parsing 2

Parse Inductives with MetaCoq

```
Inductive tm :=
| app : tm → tm → tm
| lam : ty → (tm ↦ tm) → tm
```

- Need to know which inductive types are mutually inductive
- Pairs instead of negative occurrence

```
Notation "A ↦ B" := A * B
```

State Handling

Problem: AST uses de Bruijn indices and is Globalized

```
a = b ≅
(tApp (tInd {| inductive_mind :=
                (MPfile ["Logic"; "Init"; "Coq"], "eq"); ... |}
    [ tInd {| inductive_mind :=
               (MPfile ["Datatypes"; "Init"; "Coq"], "ℕ"); ... |}
    ; tRel 1
    ; tRel 0 ]))
```

Solution: Environments

$$dbmap : string \to \mathtt{M}\ \mathbb{N}$$

$$env : string \to \mathtt{M}\ term$$

MetaCoq Problems

Conceptual

- Defining tactics in MetaCoq
- Serialization of generated lemmas

Practical

- Term explosions due to laziness
- Debugging Gallina/MetaCoq programs

Extensions

Remove Functional Extensionality

Setoid rewriting should be possible

```
Lemma subst_tm_ext (s: tm) (σ τ: ℕ → tm):
  (∀ n, σ n = τ n) →
    s[σ] = s[τ].
```

New Lemmas

```
(* evaluating predicates on all free variables *)
Lemma allfv_term_impl (s: tm) (p q: ℕ → ℙ):
  (∀ n, p n → q n) →
    allfv_term p s → allfv_term q s.
```

Summary

	Original Autosubst	Autosubst OCaml	Autosubst MetaCoq
👍	fully featured	Coq library takes care of AST/printing	used from within Coq
👎	generated code less readable custom AST	coupled to Coq version no modular syntax	worse discoverability no modular syntax
Input	lang.sig 🗎	lang.sig 🗎	`Definition lang :=` </> `Inductive lang :=` </>
Output	lang.v 🗎	lang.v 🗎	Side effect that registers lemmas in Coq

Questions?

Bibliography I

Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. (1991).
Explicit substitutions.
Journal of functional programming, 1(4):375–416.

Barendregt, H. P. (1984).
The-calculus, its syntax and semantics.
Studies in Logic, 103.

De Bruijn, N. G. (1972).
Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem.
In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier.

Bibliography II

Herbelin, H. and Lee, G.
Formalizing logical metatheory: Semantical cutelimination using kripke models for first-order predicate logic.

Laurent, O. (2021).
An anti-locally-nameless approach to formalizing quantifiers.
In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 300–312.

McBride, C. (2018).
Everybody's got to be somewhere.
arXiv preprint arXiv:1807.04085.

Bibliography III

McBride, C. and McKinna, J. (2004).
Functional pearl: i am not a number–i am a free variable.
In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 1–9.

Pfenning, F. and Elliott, C. (1988).
Higher-order abstract syntax.
ACM sigplan notices, 23(7):199–208.

Pit-Claudel, C. and Bourgeat, T.
An experience report on writing usable dsls in coq.

Schäfer, S., Smolka, G., and Tebbi, T. (2015).
Completeness and decidability of de bruijn substitution algebra in coq.
In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 67–73.

Bibliography IV

Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., and Winterhalter, T. (2019).
Coq coq correct! verification of type checking and erasure for coq, in coq.
Proceedings of the ACM on Programming Languages, 4(POPL):1–28.

Stark, K. (2019).
Mechanising syntax with binders in coq.

Binders & Substitutions

Variables as Strings

```
| lambda : string → term → term
```

Natural but need to avoid capture $(\lambda x.y)[y \mapsto x] = \lambda x.x$
α-equivalence often just assumed

Single Substitutions

```
Lemma swap_subst: ∀ t x1 x2 v1 v2,
    x1 ≠ x2 →
    closed v1 → closed v2 →
    t[x1 ↦ v1][x2 ↦ v2] = t[x2 ↦ v2][x1 ↦ v1].
```

Simple but need to deal with multiple substitutions

Sigma Calculus Background

Renamings & Substitutions

$$\xi : \mathbb{N} \to \mathbb{N}$$
$$\uparrow n = n + 1$$
$$\sigma : \mathbb{N} \to \mathit{term}$$

$$x[\sigma] = \sigma\ x$$
$$(t_1\ \ t_2)[\sigma] = t_1[\sigma]\ \ t_2[\sigma]$$
$$(\lambda.t)[\sigma] = \lambda.t[\Uparrow \sigma]$$
$$\Uparrow \sigma = 0 \cdot (\sigma \circ \langle \uparrow \rangle)$$

Using parallel subsitutitons

$$\begin{aligned}(\lambda.(\lambda.0\ 1\ \ 2))3 &\to (\lambda.0\ 1\ \ 2)[3 \cdot \mathit{id}] \\ &= \lambda.(0\ 1\ \ 2)[\Uparrow (3 \cdot \mathit{id})] \\ &= \lambda.(0\ 1\ \ 2)[0 \cdot (3 \cdot \mathit{id} \circ [\uparrow])] \\ &= \lambda.0\ 4\ 1\end{aligned}$$

Sigma Calculus Background

Equational Theory of λ calculus

$$s[var] = s$$
$$s[\sigma][\tau] = s[\sigma \circ [\tau]]$$
$$var \circ [\sigma] \equiv \sigma$$
$$(\sigma \circ [\tau]) \circ [\theta] \equiv \sigma \circ [\tau \circ [\theta]]$$
$$\sigma \circ [var] \equiv \sigma$$

$$id \circ f \equiv f$$
$$f \circ id \equiv f$$
$$(f \circ g) \circ h \equiv f \circ (g \circ h)$$
$$(s \cdot \sigma) \circ f \equiv (f\ s) \cdot (\sigma \circ f)$$
$$\uparrow \circ (s \cdot \sigma) \equiv \sigma$$
$$0 \cdot \uparrow \equiv id$$
$$(\sigma\ 0) \cdot (\uparrow \circ \sigma) \equiv \sigma$$

Scope Variables in AST

Original Autosubst had a dedicated AST node for scope variables

Scoped Simply Typed Lambda Calculus

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

tm n ≅ TmApp (TmId "tm") (SubstScope [TmId "n"])
     ≅ CApp (CRef "tm") [CRef "n"]
```

- Original Autosubst: don't print the scope variable nodes
- Autosubst OCaml: don't create nodes for scope variables

State Handling

Creating a Lemma

```
Definition make_newlemma : M term :=
  let xs := ["x0"; "x1"] in
  dbmap_adds xs;;
  ...
  oldlemma <- env_get "oldlemma" in
  ...
  let newlemma := build (dbmap_get "x1", oldlemma) in
  return ("newlemma", newlemma).

Definition tmNewlemma env : TemplateMonad Env :=
  (name, lemma) <- M.run make_newlemma env
  tmDefinition name lemma
  let nextenv := env ++ tmLookup name
  tmReturn nextenv.
```

Why Use Functional Extensionality

$$\forall s. s[\sigma][\tau] = s[\sigma \circ [\tau]]$$
$$[\sigma] \circ [\tau] = [\sigma \circ [\tau]]$$
$$\ldots = s[\ldots[\sigma] \circ [\tau] \ldots]$$

```
Lemma subst_tm_ext (s: tm) (σ τ: ℕ → tm):
  (∀ n, σ n = τ n) →
    s[σ] = s[τ].
```

Case Studies that (almost) use Autosubst

- Call by Push Value
- Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory
- Coq à la carte

- Did not use it because of FE:
- Trakhtenbrot's Theorem in Coq
- Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq

Extensions

Traced Syntax ([Herbelin and Lee,])

```
Inductive tm (vs : list ℕ) :=
  | var : v → (H: v ∈ vs) → tm vs
  | app : tm vs → tm vs' → tm (vs ++ vs')
  | lam : ty → tm ⇑vs → tm vs.
```

Extensions

New Lemmas

```
Lemma allfv_term_impl (s: tm) (p q: ℕ → ℙ):
  (∀ n, p n → q n) →
    allfv_term p s → allfv_term q s.

Lemma subst_tm_ext (s: tm) (σ τ: ℕ → tm):
  (∀ n, σ n = τ n) →
    s[σ] = s[τ].

Lemma ext_allfv_subst_term (s: tm) (σ τ: ℕ → tm):
  allfv_term (fun x ⇒ σ x = τ x) s →
    s[σ] = s[τ].
```