Generating Infrastructural Code for Terms with Binders using MetaCoq

Bachelor Talk 2

Author: Adrian Dapprich
Advisor: Andrej Dudenhefner

Department of Computer Science
Saarland University

22. July 2021

Motivation

Problem: Prove Metatheorems of Languages Modelled in Coq

- How to model binders and substitution

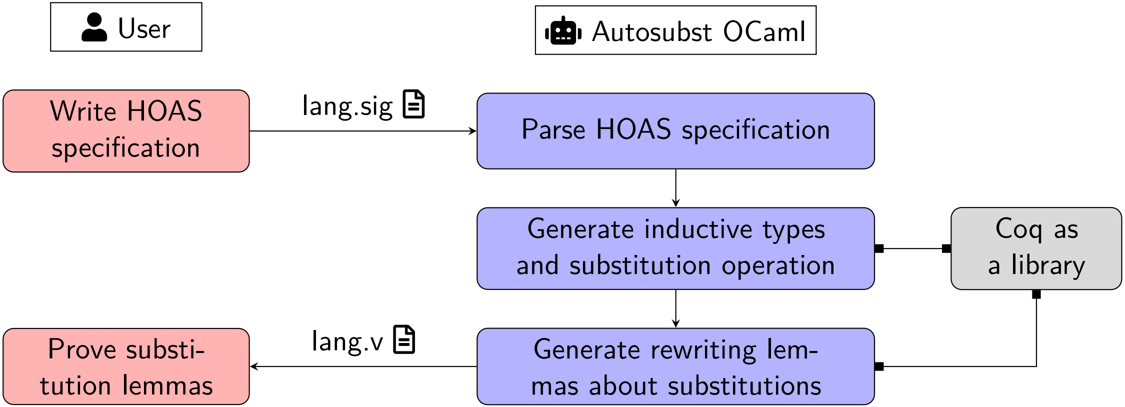
  $(\lambda x.t)v \succ_\beta t[x \mapsto v]$
- How to solve subtitution equations

  $\mathtt{s}\,[\sigma] \stackrel{?}{=} \mathtt{t}\,[\tau]$

Solution: Autosubst (Dissertation of Kathrin Stark [Stark, 2019])

- De Bruijn indices
- Based on sigma calculus [Abadi et al., 1991]
- Provides `asimpl` tactic to solve substitution equations

Workflow: Autosubst OCaml



Code Generation

- Variations of old lemmas supporting funext-free `asimpl` are generated
- Some original lemmas are optionally generated

Automation Generation

- Tactics can be constructed with tactic AST from Coq implementation (but Ltac commands can not)

```
let tac = repeat_ (first_ [ progress_ (setoid_rewrite_ asimpl_rewrite_lemma)
                          ; progress_ (unfold_ asimpl_unfold_functions)
                          ; progress_ (cbn_ asimpl_cbn_functions)
                          ... ]

Ltac asimpl := ...
```

- Typeclasses and instances can be constructed from the command & term ASTs

Code Generation

- Basic lemmas are generated
  (unscoped, functor-less and non-variadic syntax)
- wellscoped, functor and variadic syntax are straightforward extensions

Problems

- Implicit arguments
- Shadowing
- Recursive functions
- De Buijn indices

Implicit Arguments

Problem

Which arguments are implicit is not part of MetaCoq AST

Workaround

Pass "holes" (underscores in concrete syntax)

```
tmTypedDefinition "myList" hole (tApp <% @cons %> [hole; <% 0 %>; <% [] %>])
(* ⇒ myList : ?T := cons ?T0 0 [] *)
(* ⇒ mylist : list ℕ := [0] *)
```

Recursive Functions

Problem: Porting Recursive Functions to MetaCoq

Are all 23 recursive functions from OCaml terminating and implementable in Coq?

Answer: Yes

- Most are structurally recursive helper functions on lists
- Some use recursion nested in lists like rose trees
- One uses well-founded recursion with an agenda argument can be reformulated to use a fold

De Bruin Indices

Problem: Programming with De Bruin Indices is Hard

```
Fixpoint even (n: ℕ) :=
  match n with
  | S n ⇒ negb (odd n) | ...
with odd (n: ℕ) :=
  match n with
  | S n ⇒ negb (even n) | ...
```

```
tFix [
  tLam (tCase 0
    [ tLam (negb (2 0)); ... ]);
  tLam (tCase 0
    [ tLam (negb (3 0)); ... ])
  ]
```

Solution: Environments

Function `env : string` $\to \mathbb{N}$ that is updated when constructing a term below a binder

De Bruin Indices

Problem: Managing Environments is Hard

- Need to know the context before constructing a term

```
let smallerTerm = tApp (env "even") [env "n"] in
let t = buildBiggerTerm smallerTerm in
```

- Monadic functions are pervasive and you have to worry about order of execution

```
let mSmallerTerm = mApp (mEnv "even") [mEnv "n"] in
let t = mBuildBiggerTerm mSmallerTerm in
```

Solution: Custom AST with Named Variables

Translate the named variables to deBruijn indices after the whole term is built

asimpl With Setoid Rewriting

```
Lemma extequal : ∀ f g x, f x = g x.
```

Goal: Solve a Substitution Equation

```
∀ (s t: tm) f g h,
    s[t .: (h >> f)] = s[t .: (h >> g))].
```

Need morphisms for instantiation, scons and function composition†

```
Instance subst_morphism :
  Proper (pointwise_relation _ eq ⇒ eq ⇒ eq) (@subst_tm).
```

asimpl With Setoid Rewriting

Problems

- Setoid rewrite requires exact match (before typeclass resolution begins)

```
H : ∀ x, f x = g x

s[h >> f] = s[h >> g] (* Tactic failure: nothing to rewrite *)
s[fun x ⇒ f (h x)] = s[fun x ⇒ g (h x)]
```

- Morphisms are hard to get right
  Need one for all user-defined types with term indices (e.g. $\Gamma \vdash s[\sigma] : t$)
  even harder if language has nested recursion (e.g. record types)
- Slower

Allfv Lemmas

Existing infrastructure works well for this kind of new lemmas
Handle variable case, combination of recursive calls and lifting

```
Fixpoint subst (σ : ℕ → tm) (s : tm) :=
  match s with
| var s0 ⇒ σ s0
| app s0 s1 ⇒
      app (subst σ s0) (subst σ s1)
| lam s0 s1 ⇒ lam s0 (subst (⇑ σ) s1)
  end.
```

```
Fixpoint allfv (p: ℕ → ℙ) (s: tm) :=
  match s with
  | var x ⇒ p x
  | app s0 s1 ⇒
        allfv p s0 ∧ allfv p s1
  | lam s0 s1 ⇒ True ∧ allfv (⇑ p) s1
  end.
```

Code Statistics

LoC

	Haskell	OCaml	MetaCoq
code	2636	3285	2828
comments	310	437	-

Timings

asimpl

Comparing compilation times of a large case study (containing a.o. POPLmark[Aydemir et al., 2005])

functional extensionality	setoid-rewriting
111.7 seconds	412.0 seconds

Bugfixes

- Original Autosubst
  - Some printed notations
  - Unparseable substitution operation generated
  - Missing `{struct s}` annotation caused slowdonws
- Coq
  - Printing of "Existing Instances" command

Feature Table

	Autosubst OCaml	Autosubst MetaCoq	New `asimpl`
done	parsing basic lemmas lemmas for new `asimpl` tactics	parsing basic lemmas†	define lemmas morphisms proof-of-concept
todo	allfv lemmas full documentation publish	allfv lemmas full documentation publish †syntax extensions lemmas for new `asimpl` tactics	fix bugs

Maybe Todo

Faster PoC for `asimpl`, traced syntax, Autosubst webservice

Bibliography I

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

- Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. (2005).
  Mechanized metatheory for the masses: the p opl m ark challenge.
  In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer.

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

Bibliography II

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.

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.

Shadowing

Problem: Shadow existing constants

When dynamically defining new constants from a meta-program

```
Inductive ty := ... | all : ty → ty.

(* all : reductionStrategy *)
tmUnquoteInductive "tm" (Some all) ind;;
(* all : ty → ty *)
tmDefinition "mydef" (Some all) term;; (* fails *)
```

Solution

Put user generated code into a module

Custom AST

```
Inductive term :=
| tRel : ℕ → term
| tProd : string → term → term → term
| tLambda : string → term → term → term
| tApp : term → term → term
| ...
```

```
Inductive nterm :=
| nRef : string → nterm
| nTerm : term → nterm
| nProd : string → nterm → nterm →
    nterm
| nLambda : string → nterm → nterm →
    nterm
| nApp : nterm → nterm → nterm
| ...
```

Faster Alternative to Setoid Rewriting

Do Setoid Rewriting Backwards

- setoid-rewriting: given an equality, find a path of morphisms that lead to being able to rewrite with that equality
- idea: because our rewriting is pretty regular, start applying morphisms as long as subterms are not equal and apply the rewrite lemmas if we can't decompose terms further
- works well on subsitution equations $s[\sigma] \stackrel{?}{=} t[\tau]$
- does not work on normalizing single terms $s[\sigma]$

Allfv use cases

- Closedness check with constant $\bot$ predicate
- Check if a term is wellscoped
- If type function instead of predicate, collect free variables in list
- Prove two substitutions equal if they agree only on the free variables

Allfv Lemmas

```
Fixpoint idSubst (σ : ℕ → tm)
  (Eq : ∀ x, σ x = var x) (s : tm) :
    subst σ s = s :=
  match s with
  | var s0 ⇒ Eq s0
  | app s0 s1 ⇒
      congr_app (idSubst σ Eq s0)
                (idSubst σ Eq s1)
  | lam s0 s1 ⇒
      congr_lam s0
          (idSubst (⇑ σ) (⇑ Eq) s1)
  end.
```

```
Fixpoint allfv_triv (p: ℕ → ℙ)
  (H: ∀ x, p x) (s: tm) :
    allfv p s :=
  match s with
  | var s0 ⇒ H s0
  | app s0 s1 ⇒
      conj (allfv_triv p H s0)
           (allfv_triv p H s1)
  | lam s0 s1 ⇒
      conj I
        (allfv_triv (⇑ p) (⇑ H) s1)
  end.
```