Generating Infrastructural Code for Terms with Binders using MetaCoq and OCaml

Final Bachelor Talk

Author: Adrian Dapprich
Advisor: Dr. Andrej Dudenhefner
Supervisor: Prof. Gert Smolka

Department of Computer Science
Saarland University

03. November 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 substitution equations

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

Solution: Autosubst [Schäfer et al., 2015b] & Autosubst 2 [Stark, 2019]

- De Bruijn syntax & parallel substitutions
- Based on sigma calculus [Abadi et al., 1991]
- Provides `asimpl` tactic to solve substitution equations [Schäfer et al., 2015a, Stark, 2019]

Substitutions

Parallel Substitutions

Replace all variables at once. $\sigma \; : \; \mathbb{N} \to X$

$$(_ \cdot _) : X \to (\mathbb{N} \to X) \to (\mathbb{N} \to X)$$
$$(x \cdot \sigma)\ 0 = x$$
$$(x \cdot \sigma)\ (S\ n) = \sigma\ n$$

Autosubst 2's Vector Substitutions

$$T, U \in ty \ := n \mid T \to U \mid \forall . T \qquad n \in \mathbb{N}$$
$$s, t \in tm \ := n \mid s\ t \mid s\ T \mid \lambda . s \mid \Lambda . s \qquad n \in \mathbb{N}$$

$$_[_;\ _] \;:\; tm \to (\mathbb{N} \to ty) \to (\mathbb{N} \to tm) \to tm$$
$$(s\ T)[\sigma_{ty};\ \sigma_{tm}] \;=\; s[\sigma_{ty};\ \sigma_{tm}]\ T[\sigma_{ty}]$$

Autosubst Compiler

Autosubst Compiler $\longrightarrow$

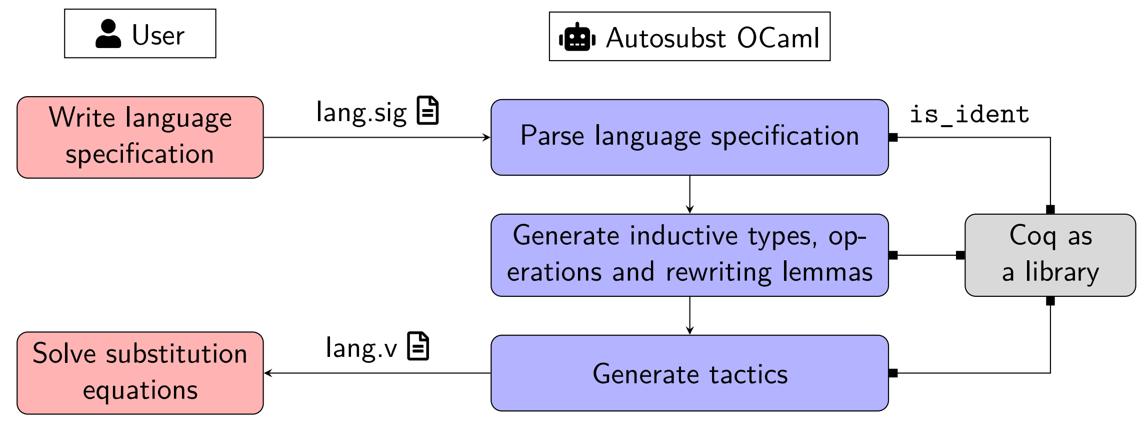
$$s, t \in tm ::= n \mid s\ t \mid \lambda.s$$

```
tm: Type

app : tm -> tm -> tm
lam : (bind tm in tm) -> tm
```

```
Inductive tm := ...
Fixpoint subst_tm := ...
Lemma idSubst_tm : ...
Ltac asimpl := ...
```

Workflow: Autosubst OCaml



Workflow: Autosubst OCaml

```
adrian@x1c: ~/programming/bachelor/autosubst-ocaml master
$ autosubst signatures/stlc.sig -o mydevelopment/stlc.v -s ucoq
done

adrian@x1c: ~/programming/bachelor/autosubst-ocaml master!
$ autosubst signatures/sysf.sig -o mydevelopment/sysf.v -s ucoq
mydevelopment/unscoped.v exists. Overwrite [y/N]:
y
mydevelopment/core.v exists. Overwrite [y/N]:
y
done

adrian@x1c: ~/programming/bachelor/autosubst-ocaml master!
$ ls mydevelopment
core.v  stlc.v  sysf.v  unscoped.v
```

Code Generation

Terms (constr_expr AST)

```
app_ f [a; b]
```

↓ pr_lconstr_expr

```
"f a b"
```

Vernacular Commands (vernac_expr AST)

```
definition_ "mydef" type body
```

↓ pr_vernac_expr

```
"Definition mydef : type := body."
```

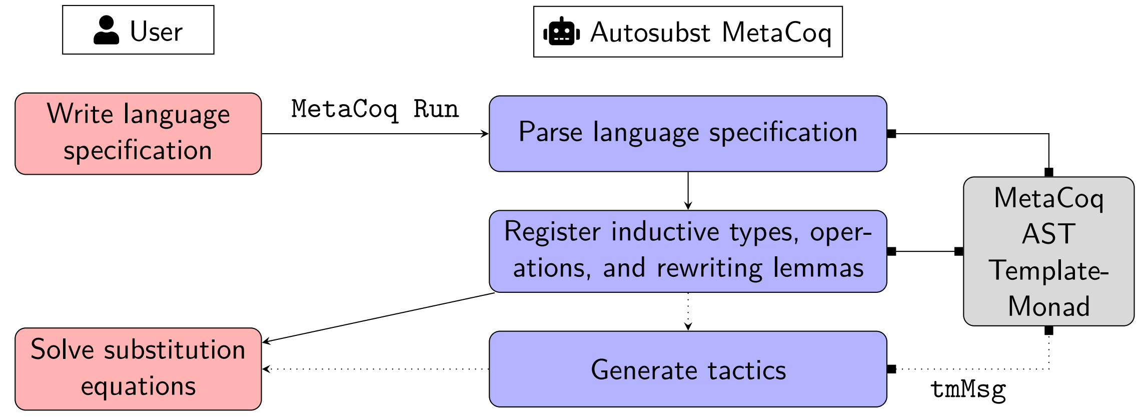
Code Generation

Tactics (raw_tactic_expr AST)

```
then_ [rewrite_ "f"; rewrite_ "g"]
          ↓ pr_tactic_ltac
   "rewrite f; rewrite g"
```

Functionality for "`Ltac mytac := ...`" has to be added.

Workflow: Autosubst MetaCoq



Workflow: Autosubst MetaCoq

```
60 (* The simply typed lambda calculus. *)
61 Definition stlc : autosubstLanguage :=
62   {| al_sorts := <{ ty : Type;
63                     tm : Type }>;
64      al_ctors := {{ Base : ty;
65                     Fun : ty → ty → ty;
66                     app : tm → tm → tm;
67                     lam : ty → (bind tm in tm) → tm }} |}.
68
69 Module stlc_unscoped.
70   MetaCoq Run Autosubst Unscoped for stlc.
71   Print ty.
72   Print tm.
73   Print compSubstSubst_tm.
74 End stlc_unscoped.
75
```

Parsing

```
<{ tm : Type }>
{{
   app : tm → tm → tm;
   lam : (bind tm in tm) → tm
}}
```

- Custom entry notations implement recursive descent parser.
- Delegate to Coq parser for identifiers.

Code Generation

Terms (term AST)

```
app_ f [a; b]
    ↓ tmUnquote
  f a b
```

Vernacular Commands (TemplateMonad functions)

```
definition_ "mydef" type_q body_q
        ↓ tmTypedDefinition
 Definition mydef : type := body.
```

Code Generation

Tactics

Not supported by MetaCoq. Therefore, ad-hoc AST and printer.

```
then_ [rewrite_ "f"; rewrite_ "g"]
        | pr_tactic
        ↓
"rewrite f; rewrite g"
```

`TemplateMonad`'s `tmMsg` only displays strings.

Challenges & Problems

- De Bruijn indices
- Globalized references
- Implicit function arguments

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)); ... ])
  ]
```

Workaround: Custom AST with Named Variables

Translate the named variables to de Bruijn indices after the whole term is built.

Implicit Arguments

Problem

Which arguments are implicit is not part of MetaCoq AST.

Workaround: Pass `holes`

`hole` is underscore in concrete syntax. Turns into evar during unquoting.

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

But adds complexity.

Solution: `Arguments` command for `TemplateMonad`

Add a `tmArguments` constructor to the `TemplateMonad`.
We have a proof-of-concept that can declare arguments as maximally implicit.

Extensions

- `asimpl` without functional extensionality axiom
- Explore new lemma generation
- Explore new syntax mode

asimpl With Functional Extensionality

```
Axiom funext : ∀ X Y (f g: X → Y) x, f x = g x → f = g.
Lemma extequal : ∀ x, f x = g x.
```

Goal: Solve a Substitution Equation

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

Easy rewrite with `funext`.

asimpl With Setoid Rewriting

```
Lemma pointwise_equal : pointwise_relation _ eq f g.
```

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.
```

Code Statistics

- Major LoC growth due to formatting
- Still lacking comments

Per Project

	Haskell	OCaml	MetaCoq
code	2636	3804	4273
comments	310	643	747
time effort		ca. 150h	ca. 100h

Contributions

- Existing from ACP
  - Basic implementation of Autosubst OCaml without tactics/notations
- Thesis
  - Fully functional implementation of Autosubst OCaml
  - Basic implementation of Autosubst MetaCoq[1]
  - Axiom-less asimpl implemented in Autosubst OCaml
  - Fixes in other projects
  - Experimental generation of some `allfv` lemmas
- Potential Future Work
  - Determine theory behind proof terms of `allfv` lemmas
  - Implement and compare formalizations of traced syntax [Herbelin and Lee, 2011]
  - Add more documentation

[1] still some lemmas missing, which prevents tactics. But easy to add.

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 poplmark challenge.
In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer.

Herbelin, H. and Lee, G. (2011).
Formalizing logical metatheory: Semantical cut elimination using kripke models for first-order predicate logic.

Bibliography II

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

Schäfer, S., Tebbi, T., and Smolka, G. (2015b).
Autosubst: Reasoning with de bruijn terms and parallel substitutions.
In *International Conference on Interactive Theorem Proving*, pages 359–374. Springer.

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.

Bibliography III

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

asimpl With Setoid Rewriting

Problems

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

  Therefore, we use the pointwise_relation abstraction

```
H : ∀ x, f x = g x

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

- Morphisms are hard to get right
  Need one for 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

Fixes

- Original Autosubst
  - Some printed notations
  - Unparseable substitution operation generated
  - Missing `{struct s}` annotation caused slowdonws
- Coq
  - Printing of `Existing Instances` command
- `monadic` library
  - Monadic filter function typo

Globalized References & the `TemplateMonad`

References are Fully-Qualified

```
MetaCoq Test Quote plus.
(* ⇒ (tConst (MPfile ["Nat"; "Init"; "Coq"], "add") []) *)
```

Solution: Build References with `tmCurrentModPath`

```
currentModPath <- tmCurrentModPath tt;;
...
tmTypedDefinition "lemma0" ...;;
...
let ref = tConst (currentModPath, "lemma0")
```

Not possible to change modules with `TemplateMonad` so all definitions are in the current module.

Input Syntax

ident	::= <Coq identifier>
ct	::= <Coq S−expression>
lang	::= decl+
decl	::= sortDecl
	| functorDecl
	| constructorDecl
sortDecl	::= $ident_s$ ':' 'Type'
functorDecl	::= $ident_F$ ':' 'Functor'
constrDecl	::= ident ':' (arg '−>')* $ident_s$
	| ident params ':' (arg '−>')* $ident_s$
params	::= '(' param (',' param)* ')'
param	::= ident ':' ct
arg	::= arghead
	| 'bind' binder (',' binder)* 'in' arghead
binder	::= $ident_s$
	| '<' ident ',' $ident_s$ '>'
arghead	::= $ident_s$
	| $ident_F$ ct arghead+

Traced Syntax

- Alternative treatment of de Bruijn indices
- Annotate each term with list of free variables that can occur
- Variable constructor takes a `finL vs`

```
Inductive wrapNat (n: ℕ) := wN : wrapNat n.

Fixpoint finL (l: list ℕ) : Type :=
  match l with
  | [] ⇒ False
  | n :: l ⇒ wrapNat n + finL l
  end.
```

Traced Syntax

```
Definition var_zero : ∀ vs, finL (0::vs)
Definition shift : finL vs → finL (0::
    vs)
```

- Close to scoped syntax
- Very easy to generate code for

```
Definition var_zero : finL [0]
Definition shift : finL vs → finL (map
    S vs)
```

- More information in type
- Harder to generate code for because sometimes we have to remove the mapping

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
| nConst : 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 substitution 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
- Prove two substitutions equal if they agree only on the free variables

allfv Lemmas

Existing infrastructure works well for this kind of new operation
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.
```

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.
```