Autosubst: Automation for de Bruijn Substitutions in Lean

Initial Bachelor Seminar Talk

Sarah Mameche
Advisor: Kathrin Stark
Supervisor: Prof. Dr. Gert Smolka

March 16, 2018

Overview

- Autosubst
  - Background: the $\sigma-$calculus
  - Features of Autosubst
- The Lean Theorem Prover
  - Metaprogramming
- Autosubst in Lean

Background

- *Goal*: formalize syntax with variable binders,
  e.g. untyped λ−calculus

$$s, t : \mathrm{tm} := \quad n \mid s\ t \mid \lambda\ s$$

- representation modulo α-equivalence: **de Bruijn**

$$\lambda f\ x. f\ (\lambda x. f\ x) \quad \longrightarrow \quad \lambda. \lambda. 1\ (\lambda. 2\ 0)$$

- *Example:* β−reduction

$$(\lambda x.s)\ t\ \triangleright s^x_t \quad \longrightarrow \quad (\lambda\ s)\ t\ \triangleright\ s[\mathbf{?}]$$

The σ-calculus[1]

- σ-calculus: explicit representation of substitutions
- parallel **substitutions** $\sigma, \tau, \theta : \mathbb{N} \to \text{tm}$
  = term sequences $(\sigma(0), \sigma(1), \sigma(2) \ldots)$
- **renamings** $\xi, \zeta : \mathbb{N} \to \mathbb{N}$
- operations (for UTLC):

instantiation $s[\sigma]$

$$n\,[\sigma] = \sigma(n)$$
$$(s\ t)\,[\sigma] = s[\sigma]\ t[\sigma]$$
$$(\lambda\ s)\,[\sigma] = \lambda\ (s[\Uparrow \sigma])$$

$$\uparrow (x) = (x + 1)$$
$$s \cdot \sigma = (s,\ \sigma(0),\ \sigma(1),\ \ldots)$$
$$\Uparrow \sigma = 0 \cdot (\sigma \circ \uparrow)$$

[1] [Abadi, Cardelli, Curien, Lévy 91]

The σ-calculus[1] – Example

Example: β-reduction $(\lambda\, s)\; t \;\triangleright\; s[t \cdot \texttt{ids}]$
and **substitutivity**

$$s_1 \triangleright s_2 \;\rightarrow\; s_1[\sigma] \triangleright s_2[\sigma]$$

Problem: prove subgoal

$$s[t \cdot \texttt{ids}]\,[\sigma] \stackrel{?}{=} s[0 \cdot (\sigma \circ \uparrow)]\,[t[\sigma] \cdot \texttt{ids}]$$

[1][Abadi, Cardelli, Curien, Lévy 91]

The σ-calculus – Rewriting rules

- σ_{SP}-calculus[2] sound, complete[3]
- equational theory with confluent rewriting rules

examples of rewriting rules for σ_{SP}

$$(\sigma \circ \tau) \circ \theta = \sigma \circ (\tau \circ \theta) \qquad (s\ t)\,[\sigma] = s[\sigma]\ t[\sigma]$$

$$s[\mathtt{ids}] = s \qquad \sigma \circ \mathtt{ids} = \sigma$$

$\Rightarrow$ normalization of expressions can be automized

[2] [Curien, Hardin, Lévy 96]
[3] [Schäfer, Smolka, Tebbi 15]

Autosubst[4,5]

- Coq library, automatically derives
  - operations of the σ-calculus
  - lemmas for rewriting and proofs
- tactics for normalization

$$s[0 \cdot (\sigma \circ \uparrow)]\,[t[\sigma] \cdot \texttt{ids}]$$

$$\Big\downarrow \ \texttt{asimpl}$$

$$s[t[\sigma] \cdot \sigma]$$

[4] [Schäfer, Tebbi, Smolka 15]
[5] [Kaiser, Schäfer, Stark 17]

Autosubst[4,5]

- Coq library, automatically derives
  - operations of the σ-calculus
  - lemmas for rewriting and proofs
- tactics for normalization

$$s[t \cdot \mathtt{ids}][\sigma] \quad = \quad s[0 \cdot (\sigma \circ \uparrow)][t[\sigma] \cdot \mathtt{ids}]$$

autosubst

$$s[t[\sigma] \cdot \sigma]$$

[4] [Schäfer, Tebbi, Smolka 15]
[5] [Kaiser, Schäfer, Stark 17]

Autosubst[4,5]

- Coq library, automatically derives
  - operations of the σ-calculus
  - lemmas for rewriting and proofs
- tactics for normalization

```
Lemma substitutivity s₁ s₂ :
    s₁ ▷ s₂ → ∀σ, s₁[σ] ▷ s₂[σ] :=
Proof. induction 1; constructor; subst; autosubst. Qed.
```

[4] [Schäfer, Tebbi, Smolka 15]
[5] [Kaiser, Schäfer, Stark 17]

Autosubst[4,5]

- Coq library, automatically derives
  - operations of the σ-calculus
  - lemmas for rewriting and proofs
- tactics for normalization
- custom term languages with binders
  $\rightarrow$ Haskell-tool generates Coq-Code
- *Problems*: tactic execution slow, huge proof terms

[4] [Schäfer, Tebbi, Smolka 15]
[5] [Kaiser, Schäfer, Stark 17]

Lean

LEAN
THEOREM PROVER

Microsoft Research

interactive theorem proving

- fully verified results
- slow, work can be tedious

+

automated theorem proving

- efficiency
- no guarantee for correctness

Lean – Metaprogramming

approaches for writing tactics and automation:

- tactic language (LTac, MTac)
- reflection (Rtac, SSReflect)

in Lean:

- use object language: `meta`–**definitions**
  - API for procedures in C++–code base
  - relaxed termination and typing conditions
  - access internal states and representations (monad operations)

Lean – important `meta`-types

- type `tactic_state` reflects state of elaborator
  - local context (hypotheses, metavariables)
  - goal stack
- `tactic` monad
  - tactics faillible
  - sequencing, backtracking
- type `expr` reflects Lean expressions into object language

Lean – example of reflection

Example:

$(s\ t).[\sigma] = s.[\sigma]\ t.[\sigma]$

```
lemma rule1 (s t σ) :
  (app s t).[σ] = app (s.[σ]) (t.[σ])
```

```
meta def match_rule1 : tactic unit :=
  do  g::gs ← get_goals,
      target ← infer_type g,

      match target with
        |  ‘((app %%s %%t).[%%σ] = %%e) := apply ‘(@rule1)
        | _ := failed end
```

```
example :  (app t (var 0)).[ids]
    = app (t.[ids]) ((var 0).[ids]) := by match_rule1
```

Approach 1: naive rewriting

```
lemma substitutivity_h :
    s[t · ids] [σ] = s[0 · (σ ∘ ↑)] [t[σ] · ids]
    := by simp with rwlemmas
```

Lean and Autosubst

```
theorem substitutivity_h : ∀ (s t : term) (σ : subst),
s.[t..I].[σ] = (s.[var 0..(σ o↑S)].[t.[σ]..I]) :=

λ (s t : term) (σ : subst),
  eq.mpr
   (id_locked
      (eq.trans
         ((λ (a a_1 : term) (e_1 : a = a_1) (a_2 a_3 : term)
           (e_2 : a_2 = a_3), congr (congr_arg eq e_1) e_2)
            s.[t..I].[σ]
            s.[t.[σ]..σ]
            (eq.trans (subst_comp (t..I) σ s)
               ((λ (a a_1 : subst) (e_1 : a = a_1)
                 (a_2 a_3 : term) (e_2 : a_2 = a_3),
                   congr (congr_arg instantiate e_1) e_2)
                  (t..I o σ)
                  (t.[σ]..σ)
                  (eq.trans (cons_comp t I σ)
                     ((λ (a a_1 : term) (e_1 : a = a_1)
                       (a_2 a_3 : ℕ → term) (e_2 : a_2 = a_3),
                         congr (congr_arg cons e_1) e_2)
                        t.[σ]
                        t.[σ]
                        (eq.refl t.[σ])
                        (I o σ)
                        σ
                        (lcomp_I σ)))

                              ...
```

Lean and Autosubst

Approach 2:

- pattern matching on goal expression, top-down parsing

```
normalize_term :
  | (s t) = (normalize_term s) (normalize_term t)
  | (λ s) = λ (normalize_term s)
  | s[σ] = normalize_inst (normalize_term s) (normalize_subst σ)
  | n = refl n
```

Lean and Autosubst

Approach 2:

$s[t \cdot \texttt{ids}][\sigma]$

↓

`normalize_inst (normalize_term` $s[t \cdot \texttt{ids}]$`) (normalize_subst` σ`)`

subst_comp $s[t \cdot \texttt{ids}]$ σ

↓

$s[(t \cdot \texttt{ids}) \circ \sigma]$

↓

`norm_inst (norm_term s)` `(normalize_subst` $(t \cdot \texttt{ids}) \circ \sigma$`)`

...

↓

`normalize_comp (normalize_subst` $t \cdot \texttt{ids}$`) (normalize_subst` σ`)`

cons_comp

↓

...

Lean and Autosubst

Approach 2:

- pattern matching on goal expression, top-down parsing
- `meta`–function `normalize` constructs proof term from rewriting lemmas

$$s[t \cdot \texttt{ids}]\,[\sigma] \xrightarrow{\text{normal form:}} s[t[\sigma] \cdot \sigma]$$

⇒ `normalize` returns proof term with type

$$\forall\ \mathrm{s}\ \mathrm{t}\ \sigma,\ s[t \cdot \texttt{ids}]\,[\sigma] = s[t[\sigma] \cdot \sigma]$$

Lean and Autosubst

```
lemma substitutivity_h :
```
$$s[t \cdot \mathtt{ids}]\,[\sigma] = s[0 \cdot (\sigma \circ \uparrow)]\,[t[\sigma] \cdot \mathtt{ids}]$$
```
    := by normalize
```

```
theorem substitutivity_h' : ∀ (s t : term) (σ : subst),
s.[t..I].[σ] = (s.[var 0..(σ o↑S)].[t.[σ]..I]) :=

λ (s t : term) (σ : subst),
 eq.trans
   (subst_comp'
      (inst' (eq.refl s) (cons_comp'
        (cons' (inst' (eq.refl t) (eq.refl σ))
        (lcomp_I' (eq.refl I) (eq.refl σ))))))
 (eq.symm
    (subst_comp' (inst' (eq.refl s) (cons_comp'
       (cons' (zero' (eq.refl (var 0)) (inst' (eq.refl t) (eq.refl σ)))
       (assoc' (rcomp_I' (eq.refl σ) (lcomp_S' (eq.refl ↑S) (eq.refl I))))))))
```

Next steps

- generalize to custom term types
  - extend current tool to generate Lean-code for rewriting lemmas and proofs
  - vector substitutions to handle multi-sorted syntactic theories
  - generate normalize−function
- case studies

References i

Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. (1991).
Explicit substitutions.
In *Journal of Functional Programming*, volume 1(4), 375-416.

de Moura, L. M., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. (2015).
The lean theorem prover (system description).
In *CADE*, volume 9195, pages 378–388.

Ebner, G., Ullrich, S., Roesch, J., Avigad, J., and de Moura, L. (2017).
A metaprogramming framework for formal verification.
Proc. ACM Program. Lang., 1(ICFP).

References ii

Kaiser, J., Schäfer, S., and Stark, K. (2017).
Autosubst 2: Towards reasoning with multi-sorted de bruijn terms and vector substitutions.
In *LFMTP*, LFMTP.

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, CPP 2015, Mumbai, India*, pages 67–73. ACM.

Schäfer, S., Tebbi, T., and Smolka, G. (2015b).
Autosubst: Reasoning with de bruijn terms and parallel substitutions.
In Zhang, X. and Urban, C., editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI. Springer-Verlag.

The σ−calculus in full

instantiation $s.[\sigma]$

$$n\,[\sigma] = \sigma(n)$$
$$(s\ t)\,[\sigma] = s[\sigma]\ t[\sigma]$$
$$(\lambda\ s)\,[\sigma] = \lambda\ (s\,[\Uparrow\sigma])$$

composition

$$(\sigma \circ \tau)\,(x) = \sigma(x)\,[\tau]$$

identity $\quad \texttt{ids}\,(x) = x$

shift $\quad \uparrow(x) = (x+1)$

cons $\quad s \cdot \sigma = (s, \sigma(0), \sigma(1), \ldots)$

up $\quad \Uparrow\sigma = 0 \cdot (\sigma \circ \uparrow)$

β−reduction

$$(\lambda\ s)\ t\ \triangleright\ s\,[t \cdot \texttt{ids}]$$

η−reduction

$$(\lambda\ s\,[\uparrow]\ 0)\ t\ \triangleright\ s$$

Rewriting rules for σ_{SP}

axiomatic equality for σ_{SP}

$$(s\ t)[\sigma] = s[\sigma]\ t[\sigma] \qquad \texttt{ids} \circ \sigma = \sigma$$

$$(\lambda\ s)[\sigma] = \lambda\,(s[\Uparrow \sigma]) \qquad \sigma \circ \texttt{ids} = \sigma$$

$$0\,[s \cdot \sigma] = s \qquad (\sigma \circ \tau) \circ \theta = \sigma \circ (\tau \circ \theta)$$

$$\uparrow \circ\, (s \cdot \sigma) = \sigma \qquad (s \cdot \sigma) \circ \tau = s[\sigma] \cdot (\sigma \circ \tau)$$

$$s[\texttt{ids}] = s \qquad s[\sigma]\,[\tau] = s[\sigma \circ \tau]$$

$$0\,[\sigma] \cdot [\uparrow \circ\ \sigma] = \sigma \qquad 0 \cdot \uparrow\, = \texttt{ids}$$

- deductively equivalent to σ_{SP}–calculus[2] (sound, complete[3])
- convergent rewriting system

[2] [Curien, Hardin, Lévy 96]
[3] [Schäfer, Smolka, Tebbi 15]

More on the `normalize`−function

Pseudocode:

```
normalize_term :
  | (s t) = (normalize_term s) (normalize_term t)
  | (λ s) = λ (normalize_term s)
  | s[σ] = normalize_inst (normalize_term s) (normalize_subst σ)
  | n = refl n

normalize_subst :
  | s ·σ = normalize_cons (normalize_term s) (normalize_subst σ)
  | ⇑σ = normalize_up (normalize_subst σ)
  | σ∘τ = normalize_cmp (normalize_subst σ) (normalize_subst τ)
  | σ = refl σ
```

More on the `normalize`−function

- subprocedures `normalize_cons`, `normalize_inst`, `normalize_comp`, `normalize_up` (mutually recursive), expect normalized arguments
- use rewriting lemmas with **assumptions**

Example:

$\sigma \circ \mathbf{ids} = \sigma$

```
lemma ids_cmp‘ {σ σ‘ τ} :
 σ = σ‘ → τ = ids → σ ∘ τ = σ‘
```

```
normalize_cmp σ τ: //arguments: normalization-proof terms
                  // e.g.(σ1 = σnorm), (τ1 = τnorm)
      if (is_ids τ) then ids_cmp‘ σ τ
      else ... // other rewriting rules
```

More on the `normalize`–function

Example:

- modified rewriting lemma

$(s\ t).[\sigma] = s.[\sigma]\ t.[\sigma]$

```
lemma rule1‘ {s t σ} :
 s.[σ] = sn → t.[σ] = tn
 → (app s t).[σ] = app (sn) (tn)
```

- goal expression:

```
s,t,σ ⊢ app s t.[σ] = app s.[σ] t.[σ]
```

- `normalize` produces proof term:

```
λ s t σ, rule1’(inst’ (refl s) (refl σ))
   (inst’ (refl t) (refl σ))
```