Sulfur: Substitution Generation in Rocq using a Logical Framework

Mathis Bouverot-Dupuis (ENS Paris, Inria Saclay)
Théo Winterhalter (Inria Saclay)

March 2025 - July 2025

Simply typed lambda calculus in Rocq

with de Bruijn indices & parallel substitutions

```
Inductive ty :=
| base
| arr (A B : ty).

Inductive tm :=
| var (idx : nat)
| app (t u : tm)
| lam (T : ty) (t : tm).

Definition id : tm :=
  lam base (var 0).

(** One-step beta-reduction. *)
Inductive red : tm -> tm -> Prop.
```

```
Definition rup (r : nat -> nat) (idx : nat) : nat :=
  match idx with
  | 0 => 0
  | S idx => S (r idx)
  end.

Fixpoint rename (r : nat -> nat) (t : tm) : tm :=
  match t with
  | var idx => var (r idx)
  | app t u => app (rename r t) (rename r u)
  | lam T t => lam T (rename (rup r) t)
  end.
Definition sup (s : nat -> tm) (idx : nat) : tm :=
  match idx with
  | 0 => var 0
  | S idx => rename S (s idx)
  end.

Fixpoint substitute (s : nat -> tm) (t : tm) : tm :=
  match t with
  | var idx => s idx
  | app t u => app (substitute s t) (substitute s u)
  | lam T t => lam T (substitute (sup s) t)
  end.
```

Substitutions are tedious

1. **Writing** the substitution & renaming functions is tedious.
2. **Proving lemmas** about substitution is tedious.

```
Lemma subst_assoc (t : tm) (s1 s2 : nat -> tm) :
  t[s1][s2] = t[s1 >> s2].
```

3. **Applying lemmas** about substitution is tedious.
   E.g. for Church-Rosser on STLC one needs to prove:

```
  t1[sup s][t2[s] . sid] = t1[t2 . sid][s]
```

   which follows from basic lemmas about substitution.

Substitutions are complex

What about languages more complex than STLC, e.g. system F ?

We need to substitute in terms and in types:

```
Inductive ty :=
| ty_var (idx : nat)
| arr (A B : ty)
| all (A : ty).

Inductive tm :=
| tm_var (idx : nat)
| app (t u : tm)
| tapp (t : tm) (T : ty)
| lam (T : ty) (t : tm)
| tlam (t : tm).
```

```
Fixpoint subst_ty (s : nat -> ty) (T : ty) : ty := ...

Fixpoint subst_tm (sty : nat -> ty) (stm : nat -> tm)
  (t : tm ) : tm := ...

Lemma subst_ty_assoc s1 s2 T :
  T[s1][s2] = T[s1 >> s2].

Lemma subst_tm_assoc sty1 sty2 stm1 stm2 t :
  t[sty1, stm1][sty2, stm2] =
  t[sty1 >> sty2, stm1 >> stm2].
```

Real-world projects can use many sorts:
Syntactic Effectful Realizability in Higher-Order Logic (Cohen, Grunfeld, Kirst, Miquey) studies a language with 7 sorts. This means 7 versions of each substitution function & lemma!

Autosubst 2

Autosubst 2 : the good

Many research projects try to automate dealing with substitutions: one of the most succesful is Autosubst 2.

System F example:

```
ty : Type
tm : Type

arr : ty -> ty -> ty
all : (bind ty in ty) -> ty

app : tm -> tm -> tm
tapp : tm -> ty -> tm
lam : ty -> (bind tm in tm) -> tm
tlam : (bind ty in tm) -> tm
```

Autosubst will:

1. Generate the substitution functions.
2. Prove basic lemmas about substitution.
3. Provide a tactic **`asimpl`** which simplifies expressions using substitution lemmas.

Autosubst 2 : the bad

Cumbersome workflow: external code generator which generates Rocq .v files.

STLC.sig

```
ty : Type
tm : Type

base : ty
arr : ty -> ty -> ty

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

STLC.v

```
Inductive ty := ...
Inductive tm := ...

Definition substitute :
  (nat -> tm) -> tm -> tm.

Lemma subst_assoc t s1 s2 :
  t[s1][s2] = t[s1 >> s2].

Ltac asimpl := ...
```

Hard to extend: Autosubst 2 relies heavily on Rocq's OCaml API.

Autosubst 2 : the ugly

`asimpl` is extremely slow. On Théo Winterhalter's ghost-reflection development: more than 3/4 of total type-checking time!

```
Ltac asimpl :=
  repeat (first
    [ progress setoid_rewrite substSubst_term_pointwise
    | progress setoid_rewrite substSubst_term
    | progress setoid_rewrite substRen_term_pointwise
    | ... ].
```

The full power of `setoid_rewrite` is needed because of pointwise equality:

```
Lemma scomp_assoc (s1 s2 s3 : nat -> tm) :
  s1 >> (s2 >> s3) =1 (s1 >> s2) >> s3.
```

Starting point of my internship: make `asimpl` more efficient!

Sulfur: using reflection

A reflective `asimpl` tactic

Main idea: write `asimpl` as a reflective tactic.

Example: solving the equation `t1[sup s][t2[s] . sid] = t1[t2 . sid][s]`

Using `asimpl` on the right hand side:

```
t1[t2 . sid][s]
        | reify
        v
Tsubst ?s (Tsubst (Scons ?t2 Sid) ?t1)
        | simplify
        v
Tsubst (Scons (Tsubst ?s ?t2) ?s) ?t1)
        | evaluate
        v
t1[t2[s] . s]
```

Concrete & explicit syntax

Concrete syntax

```
Inductive tm :=
| var (idx : nat)
| app (t u : tm)
| lam (T : ty) (t : tm).

Definition subst := nat -> tm.

Definition substitute :
  subst -> tm -> tm.

Definition scomp :
  subst -> subst -> subst.
```

Explicit syntax

```
Inductive term :=
| Tvar (idx : nat)
| Tctor (c : ctor) (args : list term)
| Tsubst (s : subst) (t : term)
| Tmvar (m : mvar)
| ...
with subst :=
| Sid
| Sshift
| Scomp (s1 s2 : subst)
| Smvar (s : mvar)
| ...
```

Explicit syntax corresponds to the sigma calculus:

- **Metavariables** `Tmvar/Smvar` represent concrete terms/substitutions which can't be described by the sigma calculus.
- **Explicit renamings** and **explicit naturals** are also needed (not shown).

Logical framework

Concrete syntax is different for each language (STLC, system F, etc) and generated by Sulfur using OCaml.

Explicit syntax is **parameterized by a signature** and defined once and for all.

A signature contains:

1. The set of constructors, e.g. $\{\texttt{app}, \texttt{lam}\}$.
2. For each constructor:
    - The arity (number of arguments).
    - Which arguments contain a binder (e.g. the body in `lam`).

`asimpl`: high-level picture

Input: a term $t_{\texttt{concrete}}$
Output: a term $t'_{\texttt{concrete}}$ and a proof of $t_{\texttt{concrete}} = t'_{\texttt{concrete}}$

$t_{\texttt{concrete}}$

↓ *reify*

$t_{\texttt{explicit}}$

↓ *simplify*

$t'_{\texttt{explicit}}$

↓ *evaluate*

$t'_{\texttt{concrete}}$

```
(* concrete -> explicit *)
OCaml reify : constr -> constr.

(* explicit -> concrete *)
Definition eval : env -> term -> tm.

Definition simplify : term -> term.

Lemma soundness t e :
  eval e t = eval e (simplify t).
```

Proved correct once and for all: much more efficient. No need to build (and type-check) a large proof each time `asimpl` is called.

Implemented in Rocq (mostly): much easier to extend, e.g. implement alternate simplification strategies.

asimpl: more details

Simplification implements exactly the reduction rules of sigma calculus.

Reduction rules:

```
Inductive term_red :=
| subst_subst t s1 s2 :
  Tsubst s2 (Tsubst s1 t) ==>
  Tsubst (Scomp s1 s2) t
| ...
with subst_red :=
| sid_left s :
    Scomp Sid s ==> s
| ...

(** Soundness of reduction. *)
Lemma soundness e t t' :
  t ==> t' ->
  eval e t = eval e t'.
```

Simplification function:

```
Definition term_simpl : term -> term.


Lemma simpl_red t :
  t ==> simpl_term t.


Lemma simpl_irred t :
  irreducible (simpl_term t).
```

Sulfur in action

Théo Winterhalter's ghost-reflection development studies a dependently typed calculus:

```
From Sulfur Require Import All.

Inductive mode := ...

Sulfur Generate
{{
  term : Type

  app : term -> term -> term
  lam : {{mode}} -> term -> (bind term in term) -> term
  ...
}}.

Check substitute. (* (nat -> term) -> term -> term *)

Lemma substitute_assoc t s1 s2 :
  t[s1][s2] = t[s1 >> s2].
Proof. asimpl. reflexivity. Qed.
```

Future Work

Proving completeness (future work)

A completeness theorem holds in simpler variants of sigma calculus:

```
Theorem completeness t t' :
  (forall e, eval e t = eval e t') ->
  simpl_term t = simpl_term t'
```

Intuitively, reification followed by simplification is enough to decide equality of concrete terms.

Full completeness does not hold in our case. Possible future work:

1. Prove a weaker form of completeness.
2. Perform more aggressive simplifications to recover full completeness.

Scaling to more complex languages (WIP/future work)

Multiple sorts (e.g. system F).

```
Inductive ty :=
| ...
with tm :=
| ...
with value :=
| ...
```

Lists/options in constructor arguments (e.g. n-ary applications), and in general arbitrary functors.

```
Inductive tm :=
| app (t : tm) (ts : list tm)
| ...
```

We have made serious attempts for both features but there are technical difficulties.

Recap

1. Sulfur, a tool to help dealing with de Bruijn indices and parallel substitutions.
2. Simplification is efficient and easy to extend.
3. Good basis for theoretical experiments around sigma calculus.
4. Handling multiple sorts is challenging (future work).

Code is on github (WIP)

MathisBD more renaming — cac81d4 · 2 weeks ago — 150 Commits

meetings	add meeting notes	last month
metaprog	handle functors in the generation of the signature	2 weeks ago
plugin	more renaming	2 weeks ago
test-ghost-reflection	more renaming	2 weeks ago
theories	more renaming	2 weeks ago
utils	more renaming	2 weeks ago
.gitignore	intrinsic/extrinsic experiments	4 months ago
.ocamlformat	finish proof of congr_rename	3 months ago
README.md	more renaming	2 weeks ago
dune-project	more renaming	2 weeks ago
rocq-sulfur.opam	again more renaming	2 weeks ago

https://github.com/MathisBD/rocq-sulfur