Sulfur: a Reflective Tactic for Substitution Generation

Mathis Bouverot-Dupuis Théo Winterhalter
Kathrin Stark Kenji Maillard

Formalizing binders is difficult

POPLMark challenge [1]: binders are key when formalizing programming languages.

$$
\begin{array}{rr}
t, u ::= x & \textit{variable} \\
\mid t\ u & \textit{application} \\
\mid \lambda x.\ t & \textit{abstraction}
\end{array}
$$

Summary of bindings techniques used in POPLMark solutions: [2]

Binder representation	Proposed solutions
De Bruijn	3
HOAS	2
Weak HOAS	1
Hybrid	1
Locally Nameless	3
Named Variables	1
Nested Abstract Syntax	1
Nominal	2

[1] "Mechanized metatheory for the masses: the PoplMark challenge" (Aydemir et al)
[2] Taken from the POPLMark website https://www.seas.upenn.edu/ plclub/poplmark/

Autosubst

Many research projects try to automate dealing with binders in proof assistants. One of the most successful is **Autosubst 2**. [3] [4]

Many implementations: in Ltac, Haskell, MetaRocq, and OCaml. The most widely used is the OCaml version [5].

Autosubst is used in many formalizations.

[3] "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions" (Schäfer, Tebbi, Smolka)

[4] "Autosubst 2: Reasoning with Multi-sorted de Bruijn Terms and Vector Substitutions" (Stark, Schäfer, Kaiser)

[5] https://github.com/uds-psl/autosubst-ocaml

Autosubst 2 in action

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 a term inductive, using de Bruijn indices.
2. Generate the substitution function:
   ```
   Definition substitute : (nat -> tm) -> tm -> tm.
   ```
3. Prove basic lemmas about substitution.
4. Provide a tactic `asimpl` which simplifies expressions using substitution lemmas:
   ```
   Lemma technical_lemma t1 t2 s :
     t1[sup s][t2[s] . sid] = t1[t2 . sid][s].
   Proof. asimpl. reflexivity. Qed.
   ```

asimpl is too slow

On Théo Winterhalter's **ghost-reflection** [6] development `asimpl` takes 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.
```

Sulfur = Autosubst + efficient asimpl

[6] https://github.com/TheoWinterhalter/ghost-reflection

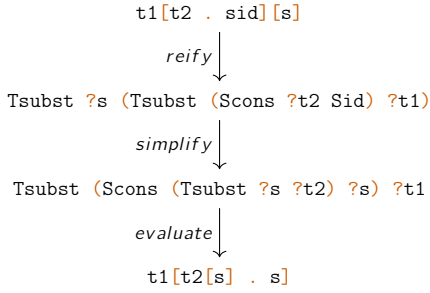
Sulfur: using reflection

A reflective asimpl tactic

How to gain performance? 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:



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** [7]:

- **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).

[7] "Explicit Substitutions" (Abadi, Cardelli, Curien, Lévy)

Concrete & explicit syntax

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:

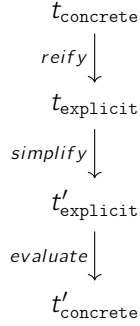
```
Inductive term (sig : signature) :=
| Tvar (idx : nat)
| Tctor (c : ctor sig) (args : list (term sig))
| Tsubst (s : subst sig) (t : term sig)
| ...
```

A signature contains:

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

asimpl: more details

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




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

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

(* explitcit -> explicit *)
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.

Future Work

Benchmarking

- No benchmarks yet, performance seems noticeably faster compared to Autosubst's asimpl (probably at least 10x).

- Many basic optimizations to implement:
  - We still use `cbv` & `cbn` instead of `vm_compute`.
  - The simplification function (written in Rocq) is very naive.
  - We need to traverse terms several times because of limitations of `rewrite_strat`.

Scaling to more complex languages

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)
| ...
```

Proving completeness

A completeness theorem holds in simpler variants of sigma calculus: [8]

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

Intuitively, reification and 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.

[8] "Completeness and Decidability of de Bruijn Substitution Algebra in Coq" (Schäfer, Smolka, Tebbi)

Recap

1. **Sulfur**, a tool to help dealing with de Bruijn indices and parallel substitutions.
2. Simplification is **efficient** and **easy to extend**.
3. Handling multiple sorts is challenging (future work).

Code is on github

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