Metaprogramming in Lean

Type Theory and Coq

Markel Zubia Aldaburu

December 10, 2021

What is metaprogramming?

- Metaprogramming: code that talks about code.
- In formal verification it can be used to define tactics.

Metaprogramming in Coq

- Coq uses Ltac.
- Untyped scripting language.

```
Ltac self_implication := intros x; assumption.
```

Metaprogramming in Lean

- Lean + `meta` keyword.

```
meta def self_implication : tactic unit :=
do intro `_,
   assumption
```

Advantages

Advantages:

1. Same language.
2. Everything in Lean's library available.
3. Write and debug the same way.

meta keyword

Two effects:

1. Access to tactic state.
2. General recursion.

expr inductive type

```
inductive expr
| var : nat → expr
| lconst : name → name → expr
| mvar : name → expr → expr
| sort : level → expr
| const : name → list level → expr
| app : expr → expr → expr
| lam : name → binfo → expr → expr → expr
| pi : name → binfo → expr → expr → expr
| elet : name → expr → expr → expr → expr
```

tactic monad

```
meta inductive result (state : Type) (a : Type)
| success : a → state → result
| exception : option (unit → format) → option pos →
                state → result

meta def interaction_monad (state : Type) (a : Type) :=
state → result state a

meta def tactic := interaction_monad tactic_state
```

tactic_state

- `tactic_state` instance of the state monad.
- Contains goals, subgoals, hypotheses, metavariables...
- `get`, `add`, `constructors_of`, `get_goals`, `set_goals`, `local_context`, `target`, ...

assumption tactic

```
meta def find : expr → list expr → tactic
    expr
| e [] := failed
| e (h :: hs) :=
    do t ← infer_type h,
        (unify e t >> return h) <|> find e hs

meta def assumption : tactic unit :=
do { ctx ← local_context,
    t ← target,
    h ← find t ctx,
    exact h }
<|> fail "assumption tactic failed"
```

rsimp

```
meta def rsimp : tactic unit :=
do ccs ← collect_implied_eqs,
    try $ simp_top_down $ \ t, do
        let root := ccs.root t,
        let t' := choose ccs root,
        p ← ccs.eqv_proof t t',
        return (t', p)
```

nano_crush

```
meta def nano_crush (depth : nat := 1) :=
do hs ← mk_relevant_lemmas,
    induct (search (rsimp' hs) depth)

meta def search (tac : tactic unit) : nat → tactic unit
| 0 := try tac >> done
| (d+1) := try tac >>
    (done <|> all_goals (split (search d)))

meta def induct (tac : tactic unit) : tactic unit :=
collect_inductive_hyps >>=
    try_list (\ e, induction' e; tac)
```

nano_crush

```
meta def split (tac : tactic unit) : tactic unit :=
collect_inductive_from_target >>=
    try_list (\ e, cases e; tac)

meta def try_list {a} (tac : a → tactic unit) :
list a → tactic unit
| [] := failed
| (e::es) := (tac e >> done) <|> try_list es
```

Example

```
attribute [simp] mul_add
lemma eeval_times (k e) :
    eeval (times k e) = k * eeval e := by nano_crush
```