White-box Automation

ITP - Brasilia - September 26, 2017

Leonardo de Moura - Microsoft Research

joint work with

Gabriel Ebner - Vienna University of Technology

Sebastian Ullrich - Karlsruhe Institute of Technology

Jared Roesch - University of Washington

Jeremy Avigad - Carnegie Mellon University

https://leanprover.github.io/papers/tactic.pdf

The Lean team

- Everybody in the previous slide, and
- Mario Carneiro (CMU),
- Johannes Hölzl (CMU),
- Floris van Doorn (CMU),
- Rob Lewis (CMU),
- Daniel Selsam (Stanford)

Former Members:

- Soonho Kong (CMU),
- Jakob von Raumer (University of Nottingham)

Many thanks to

- Cody Roux
- Georges Gonthier
- Grant Passmore
- Nikhil Swamy
- Assia Mahboubi
- Bas Spitters
- Steve Awodey
- Ulrik Buchholtz
- Tom Ball
- David Christiansen

Lean aims to bridge the gap between interactive and automated theorem proving

How are automated provers used at Microsoft?

Testing

SAGE

Pex

Software Verification

Vcc

BOOGIE

Z3 Theorem Prover

Software verification & automated provers

- Easy to use for simple properties
- Main problems:
  - Scalability issues
  - Proof stability
- in many verification projects:
  - Hyper-V
  - Ironclad & Ironfleet (https://github.com/Microsoft/Ironclad)
  - Everest (https://project-everest.github.io/)

Automated provers are mostly black-boxes

"The Strategy Challenge in SMT Solving", joint work with Grant Passmore

$$(\mathsf{check}(\neg\mathsf{diff} \vee \frac{\mathsf{atom}}{\mathsf{dim}} < \mathsf{k})\,;\,\mathsf{simplex})\,|\,\mathsf{floydwarshall}$$

$$\mathsf{simplify}\,;\,\mathsf{gaussian}\,;\,(\mathsf{modelfinder}\,|\,\mathsf{smt}(\mathsf{apcad}(\mathsf{icp})))$$

Introduction: Lean

- New open source theorem prover (and programming language)

  Soonho Kong and I started coding in the Fall of 2013

- Platform for
  - Software verification
  - Formalized Mathematics
- de Bruijn’s principle: small trusted kernel
- Dependent Type Theory
- Metaprogramming
- First official version was released at CADE 2015.

Metaprogramming

- Extend Lean using Lean
- Access Lean internals using Lean
  - Type inference
  - Unifier
  - Simplifier
  - Decision procedures
  - Type class resolution
  - …
- Proof/Program synthesis

White-box automation

APIs (in Lean) for accessing data-structures and procedures found in SMT solvers and ATPs.

The Logic Framework

- CIC-- (Calculus of Inductive Constructions)
  - - Fixpoint/Match
  - + Recursors
  - Coquand and Paulin-Mohring's Calculus of Inductive Constructions 1988
- Inductive families (P. Dybjer)
- Universe polymorphism
- Proof irrelevance

```
Type                        -- Sort
nat                         -- Constant
λ x : nat, x                -- Lambda abstraction
vector bool 3               -- Application
Π (n : nat), vector nat n -- Function Space
nat → bool                  -- Function Space (no dependency)
```

Inductive Families

```
inductive nat
| zero : nat
| succ : nat → nat

inductive tree (α : Type u)
| leaf : α → tree
| node : tree → tree → tree

inductive vector (α : Type) : nat → Type
| nil  : vector zero
| cons : Π {n : nat}, α → vector n → vector (succ n)
```

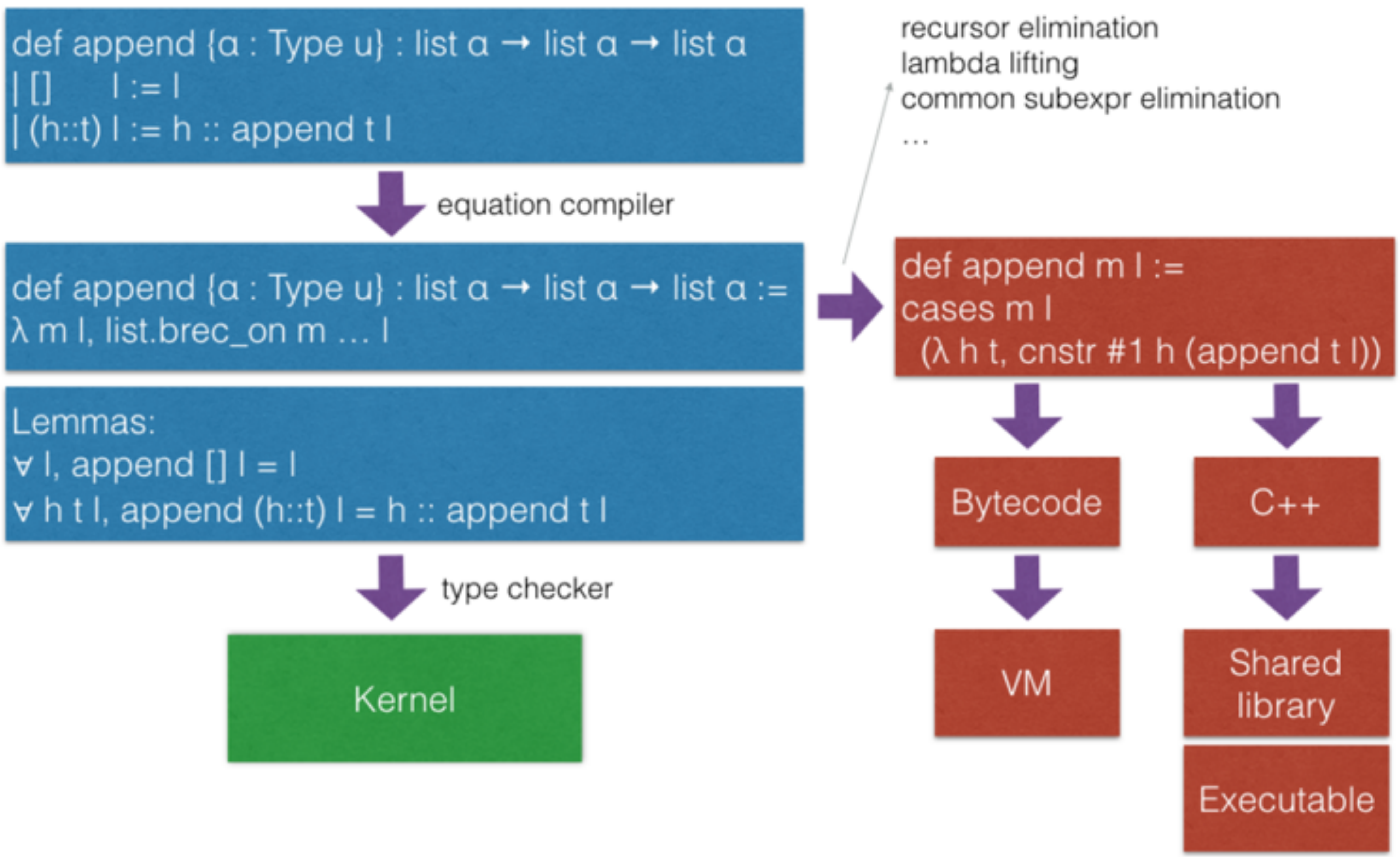
Recursive equations

- Recursors are inconvenient to use.
- Compiler from recursive equations to recursors.
- Several compilation strategies: structural, well-founded, unbounded recursion, …

```
def fib : nat → nat
| 0     := 1
| 1     := 1
| (a+2) := fib a + fib (a + 1)
```

```
def ack : nat → nat → nat
| 0     y     := y+1
| (x+1) 0     := ack x 1
| (x+1) (y+1) := ack x (ack (x+1) y)
```

Recursive equations



Mutual recursion

```
inductive term
| const : string → term
| app   : string → list term → term

mutual def num_consts, num_consts_lst
with num_consts : term → nat
| (term.const n)  := 1
| (term.app n ts) := num_consts_lst ts
with num_consts_lst : list term → nat
| []      := 0
| (t::ts) := num_consts t + num_consts_lst ts
```

Structures

```
structure point (α : Type) :=
mk :: (x : α) (y : α)

#eval point.x (point.mk 10 20)
#eval point.y (point.mk 10 20)

#eval {point . x := 10, y := 20}

def p : point nat :=
{x := 10, y := 20}

#eval p.x
#eval p.y
#eval {p with x := 1}

structure point3d (α : Type) extends point α :=
(z : α)
```

Type classes

```
class has_sizeof (α : Type u) :=
(sizeof : α → nat)

variables {α : Type u} {β : Type v}

def sizeof [has_sizeof α] : α → nat

instance : has_sizeof nat := ⟨λ a : nat, a⟩
-- ⟨...⟩ is the anonymous constructor

instance [has_sizeof α] [has_sizeof β] : has_sizeof (prod α β) :=
⟨λ p, match p with
    | (a, b) := sizeof a + sizeof b + 1
    end⟩

instance [has_sizeof α] [has_sizeof β] : has_sizeof (sum α β) :=
⟨λ s, match s with
    | inl a := sizeof a + 1
    | inr b := sizeof b + 1
    end⟩
```

Metaprogramming

```
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"

lemma simple (p q : Prop) (h₁ : p) (h₂ : q) : q :=
by assumption
```

Reflecting expressions

```
inductive level
| zero   : level
| succ   : level → level
| max    : level → level → level
| imax   : level → level → level
| param  : name → level
| mvar   : name → level
```

```
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
```

```
meta def num_args : expr → nat
| (app f a) := num_args f + 1
| e         := 0
```

Quotations

```
example : true ∧ true :=
by do apply `(and.intro trivial trivial)
```

```
example (p : Prop) : p → p ∨ false :=
by do e ← intro `h, refine ``(or.inl %%e)
```

```
meta def is_not : expr → option expr
| `(not %%a)         := some a
| `(%%a → false) := some a
| _                  := none
```

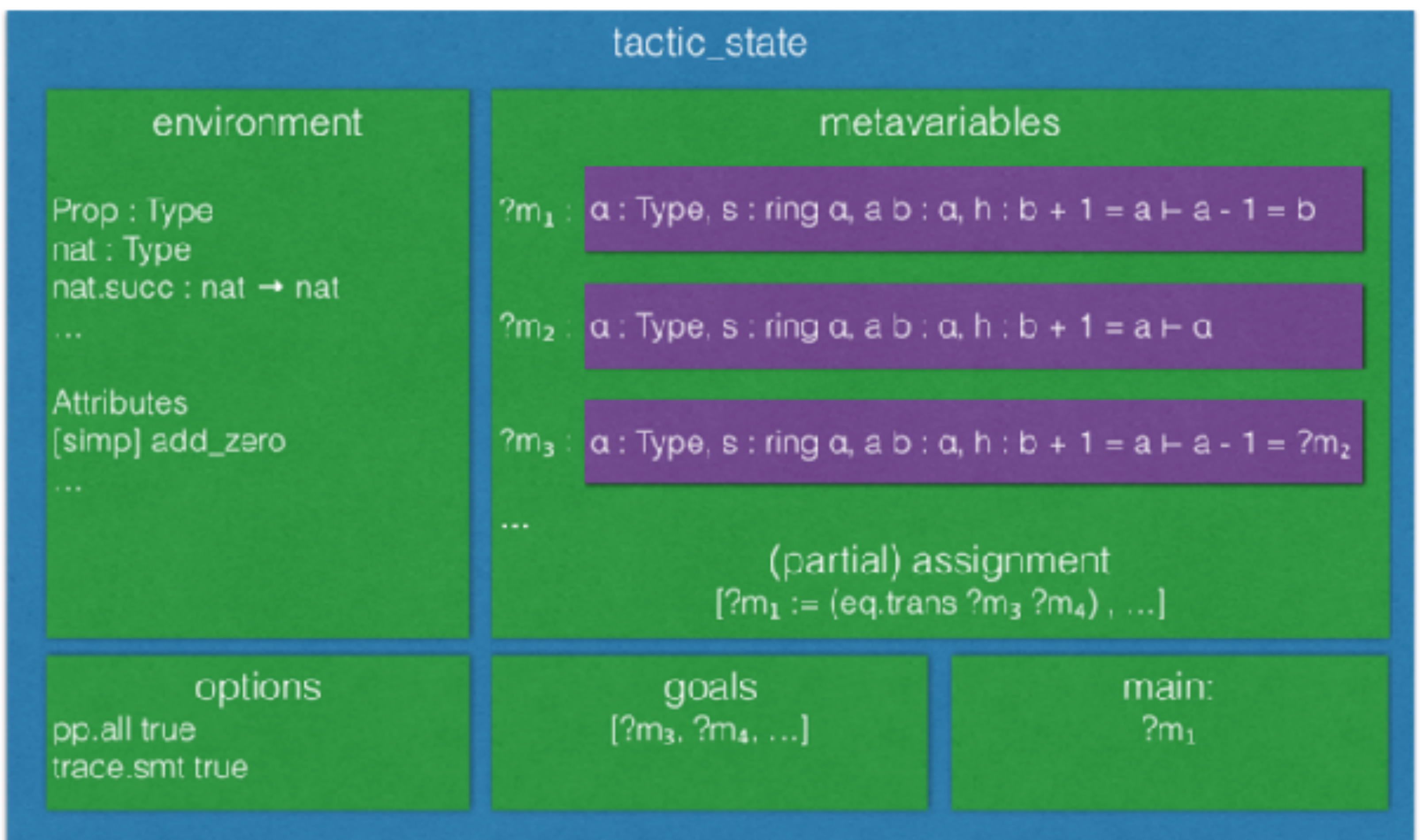
```
meta def is_not : expr → option expr
| (app (const ``not _) a)           := some a
| (pi _ _ a (const ``false _)) := some a
| _                                   := none
```

The tactic monad

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

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

meta def tactic := interaction_monad tactic_state
```



tactics

```
infer_type  : expr → tactic expr
unify       : expr → expr → tactic unit
mk_instance : expr → tactic expr
intro       : name → tactic expr
  get_goals : tactic (list expr)
  set_goals : list expr → tactic unit

meta def done : tactic unit := do [] ← get_goals, return ()
meta def target : tactic expr := do g::gs ← get_goals, infer_type g
```

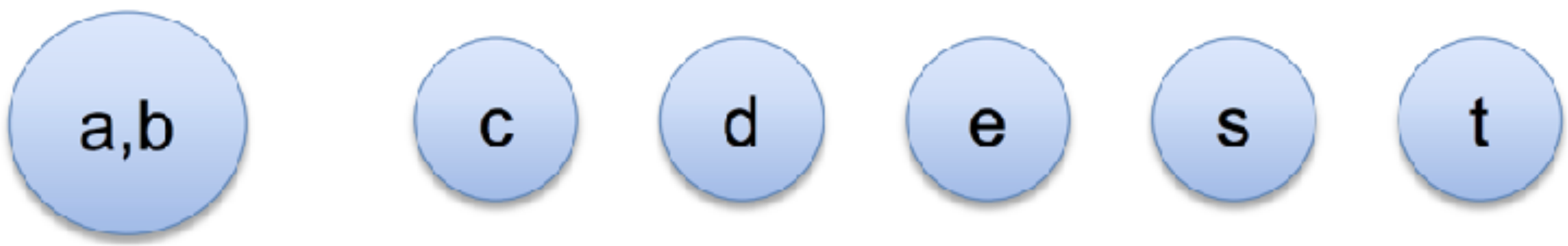
Example: congruence closure

- It is the kernel of most SMT solvers (e.g., CVC4, MathSAT, Yices and Z3).
- Efficient procedure for equality.

$$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$$

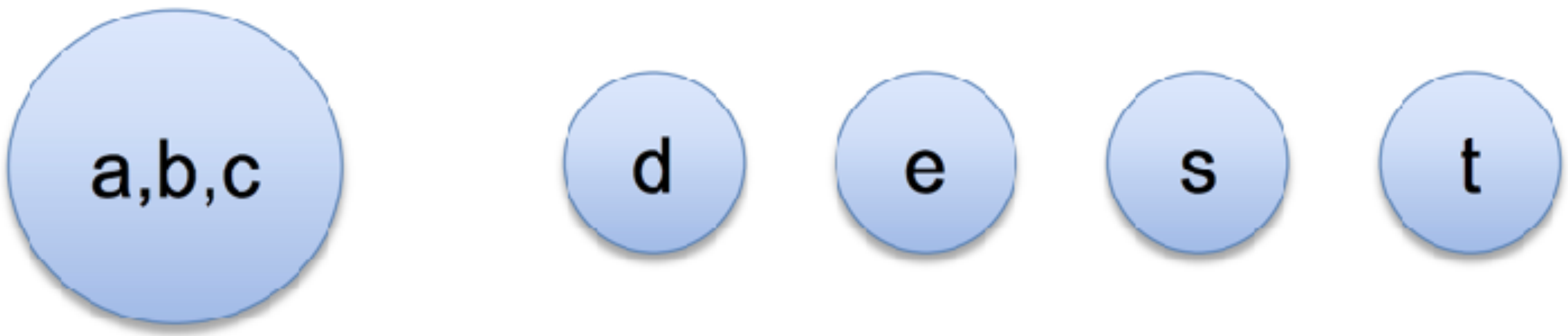
Example: congruence closure

$a = b$, $b = c$, $d = e$, $b = s$, $d = t$, $a \neq e$, $a \neq s$



Example: congruence closure

$a = b$, $b = c$, $d = e$, $b = s$, $d = t$, $a \neq e$, $a \neq s$



Example: congruence closure

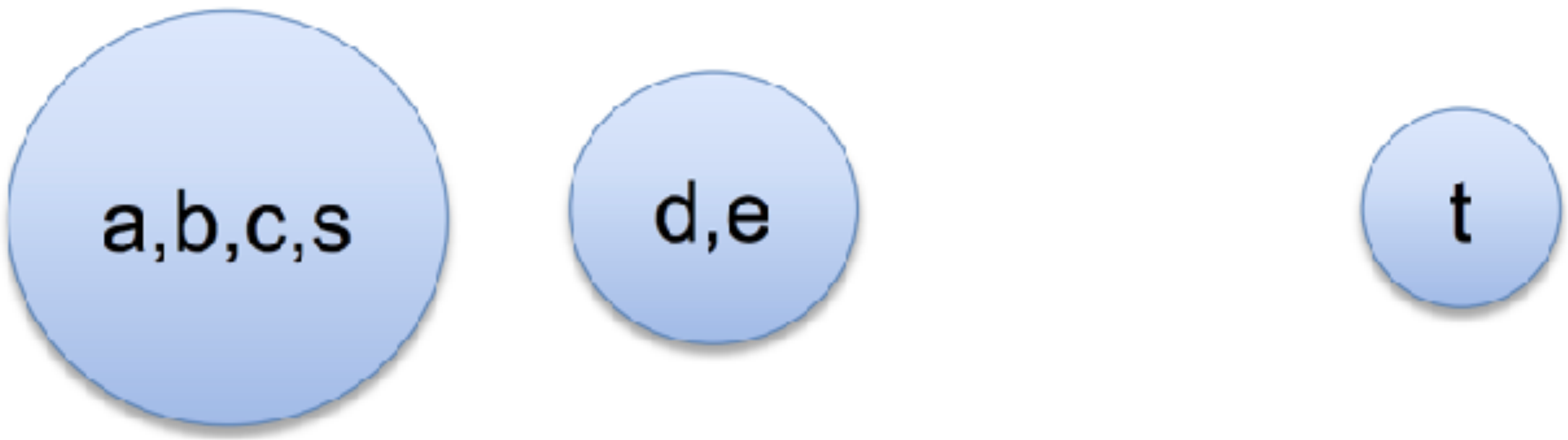
$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$

Example: congruence closure

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$

Example: congruence closure

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$



Example: congruence closure

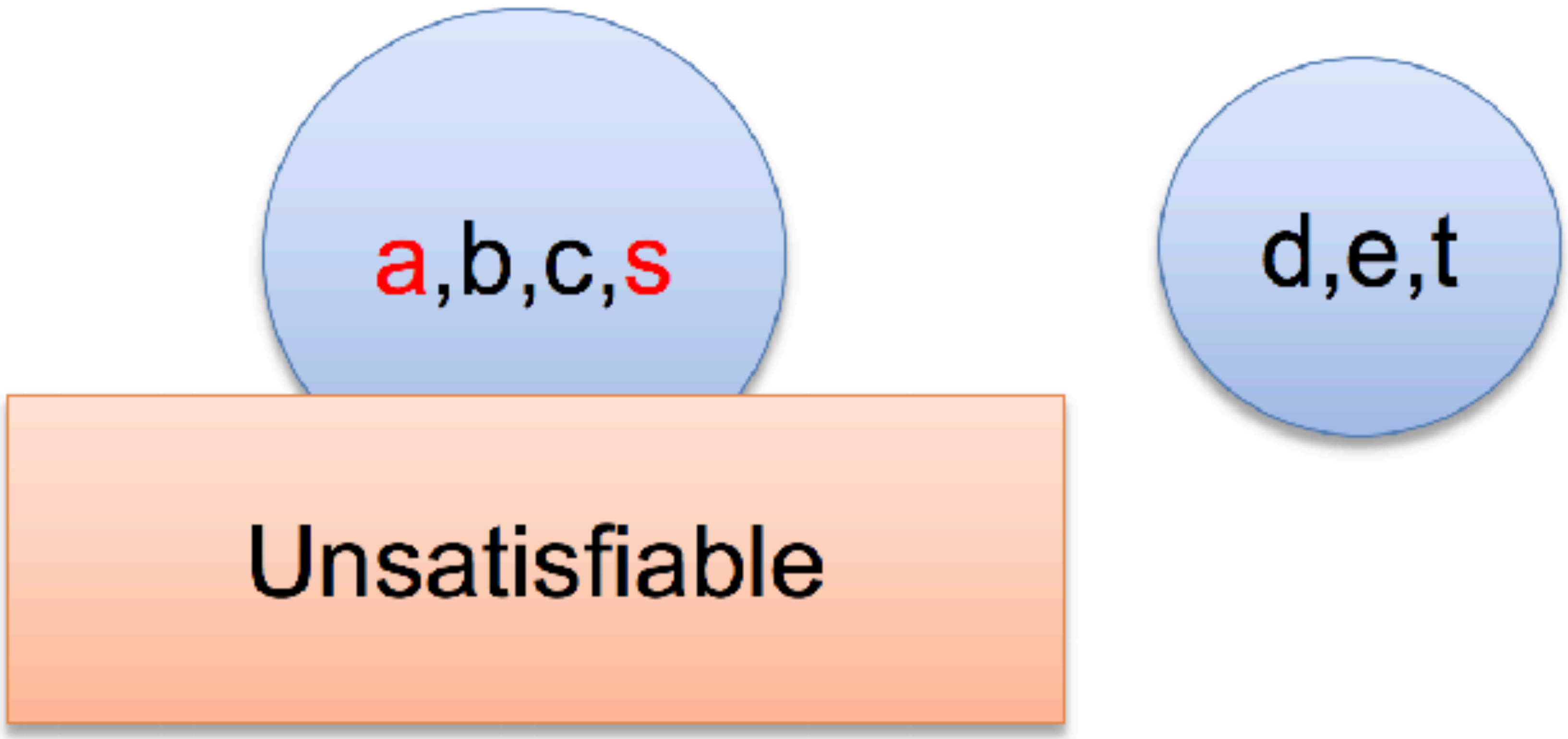
$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$

Example: congruence closure

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ a \neq e,\ a \neq s$

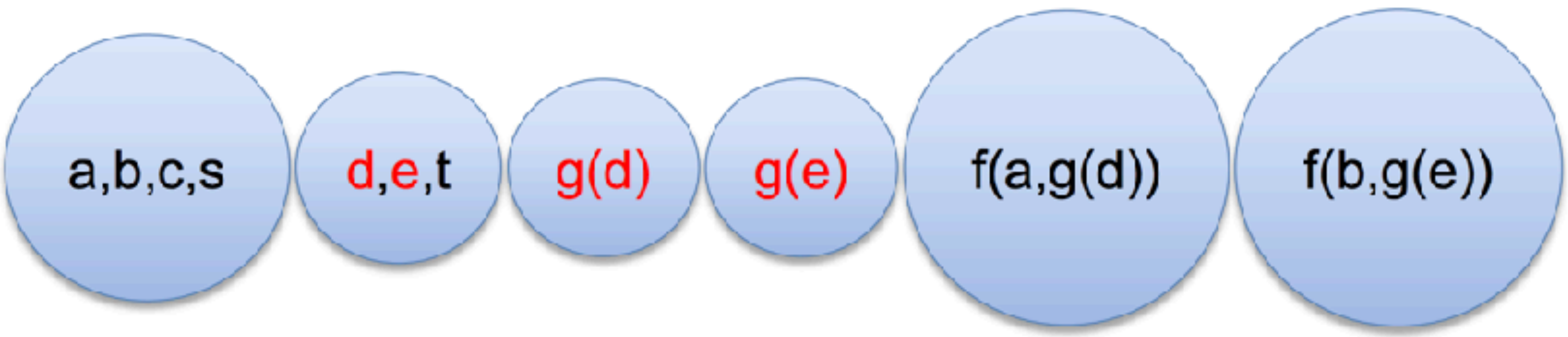
Example: congruence closure

$a = b, b = c, d = e, b = s, d = t, a \neq e, a \neq s$



Example: congruence closure

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ f(a, g(d)) \neq f(b, g(e))$



Congruence Rule:

$x_1 = y_1, \ldots, x_n = y_n$ implies $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

Example: congruence closure

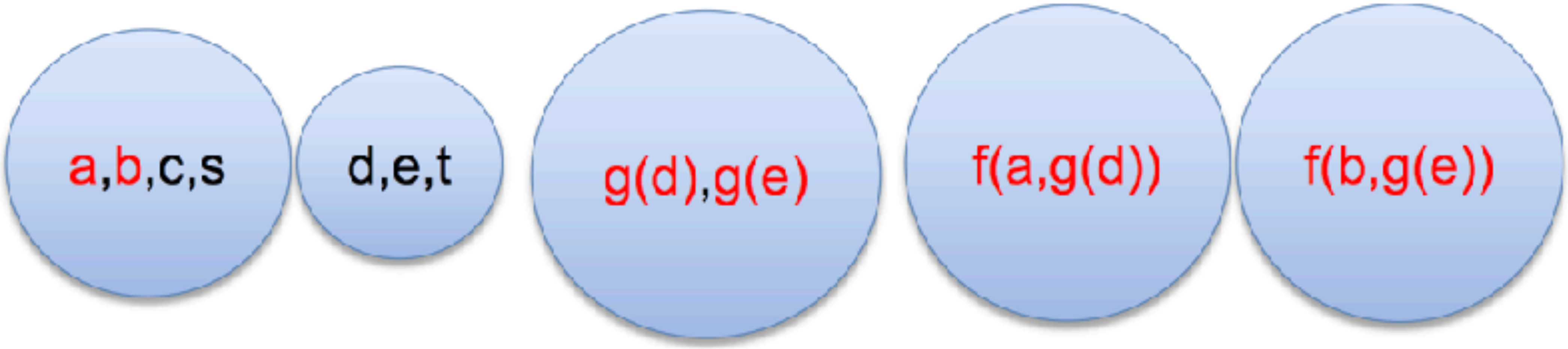
$$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ f(a, g(d)) \neq f(b, g(e))$$

Congruence Rule:

$x_1 = y_1, \ldots, x_n = y_n$ implies $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

Example: congruence closure

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ f(a, g(d)) \neq f(b, g(e))$



Congruence Rule:

$x_1 = y_1, \ldots, x_n = y_n$ implies $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

Example: congruence closure

$a = b,\ b = c,\ d = e,\ b = s,\ d = t,\ f(a, g(d)) \neq f(b, g(e))$

Congruence Rule:

$x_1 = y_1, \ldots, x_n = y_n$ implies $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

Example: congruence closure

```
/-- Congruence closure state -/
meta constant cc_state                    : Type
meta constant cc_state.mk_core            : cc_config → cc_state
/-- Create a congruence closure state object using the hypotheses in the current goal. -/
meta constant cc_state.mk_using_hs_core : cc_config → tactic cc_state
meta constant cc_state.next               : cc_state → expr → expr
meta constant cc_state.root               : cc_state → expr → expr
meta constant cc_state.mt                 : cc_state → expr → nat
meta constant cc_state.internalize        : cc_state → expr → tactic cc_state
meta constant cc_state.add                : cc_state → expr → tactic cc_state
meta constant cc_state.is_eqv             : cc_state → expr → expr → tactic bool
meta constant cc_state.is_not_eqv         : cc_state → expr → expr → tactic bool
meta constant cc_state.eqv_proof          : cc_state → expr → expr → tactic expr
meta constant cc_state.inconsistent       : cc_state → bool
/-- `proof_for cc e` constructs a proof for e if it is equivalent to true in cc_state -/
meta constant cc_state.proof_for          : cc_state → expr → tactic expr
/-- `refutation_for cc e` constructs a proof for `not e` if it is equivalent to false in cc_state -/
meta constant cc_state.refutation_for     : cc_state → expr → tactic expr
/-- If the given state is inconsistent, return a proof for false. Otherwise fail. -/
meta constant cc_state.proof_for_false    : cc_state → tactic expr
```

Example: congruence closure

```
meta def fold_eqc_core {α} (s : cc_state) (f : α → expr → α) (first : expr) : expr → α → α
| c a :=
  let new_a := f a c,
      next  := s.next c in
  if next =ₐ first then new_a
  else fold_eqc_core next new_a

meta def fold_eqc {α} (s : cc_state) (e : expr) (a : α) (f : α → expr → α) : α :=
fold_eqc_core s f e e a
```

rsimp tactic

```
constants (f : nat → nat → nat) (g : nat → nat) (p : nat → nat → Prop)
axioms (fax : ∀ x, f x x = x) (pax : ∀ x, p x x)

example (a b c : nat) (h₁ : a = g b) (h₂ : a = b) : p (f (g a) a) b :=
```

Suppose we try to simplify the target using the axiom fax and the hypotheses above

```
p (f (g b) b) b
p (f (g (g b)) (g b)) b
```

rsimp tactic

```
meta def collect_implied_eqs : tactic cc_state := ...

meta def choose (ccs : cc_state) (e : expr) : expr :=
ccs.fold_eqc e e $ λ (best_so_far curr : expr),
  if size curr < size best_so_far then curr else best_so_far
```

As in Haskell, the notation `f $ g t` is an alternative way of writing `f (g t)`

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

An example

```
constants (f : nat → nat → nat) (g : nat → nat) (p : nat → nat → Prop)
axioms (fax : ∀ x, f x x = x) (pax : ∀ x, p x x)

example (a b c : nat) (h₁ : a = g b) (h₂ : a = b) : p (f (g a) a) b :=
by rsimp; apply pax
```

```
{g b, f (g a) a, f (g a) (g a), g a, b, a}
```

```
example (a b c d : nat) : d = max c (a + b) →
                                p (max a (max d (b + a))) (max d (a + b)) :=
by intros; rsimp; apply pax
```

Extending the tactic state

```
def state_t (σ : Type) (m : Type → Type) [monad m] (α : Type) : Type :=
σ → m (α × σ)

meta constant smt_goal : Type
meta def smt_state  := list smt_goal
meta def smt_tactic := state_t smt_state tactic

meta def eblast : smt_tactic unit := repeat (ematch; try close)

meta def collect_implied_eqs : tactic cc_state :=
focus $ using_smt $ do
  add_lemmas_from_facts, eblast,
  (done; return cc_state.mk) <|> to_cc_state
```

Superposition prover

- 2200 lines of code

```
example {α} [monoid α] [has_inv α] : (∀ x : α, x * x⁻¹ = 1) →
                                      ∀ x : α, x⁻¹ * x = 1 :=
by super with mul_assoc mul_one

meta structure prover_state :=
(active passive : rb_map clause_id derived_clause)
(newly_derived : list derived_clause) (prec : list expr)
(locked : list locked_clause) (sat_solver : cdcl.state)
...
meta def prover := state_t prover_state tactic
```

dlist

```
structure dlist (α : Type u) :=
(apply     : list α → list α)
(invariant : ∀ l, apply l = apply [] ++ l)
```

```
def to_list : dlist α → list α
| ⟨xs, _⟩ := xs []
```

```
local notation `♯`:max := by abstract {intros, rsimp}
```

```
/-- `O(1)` Append dlists -/
protected def append : dlist α → dlist α → dlist α
| ⟨xs, h₁⟩ ⟨ys, h₂⟩ := ⟨xs ∘ ys, ♯⟩

instance : has_append (dlist α) :=
⟨dlist.append⟩
```

transfer tactic

- Developed by Johannes Hölzl (approx. 200 lines of code)

```
lemma to_list_append (l₁ l₂ : dlist α) : to_list (l₁ ++ l₂) = to_list l₁ ++ to_list l₂ :=
show to_list (dlist.append l₁ l₂) = to_list l₁ ++ to_list l₂, from
by cases l₁; cases l₂; simp; rsimp
```

```
protected def rel_dlist_list (d : dlist α) (l : list α) : Prop :=
to_list d = l
```

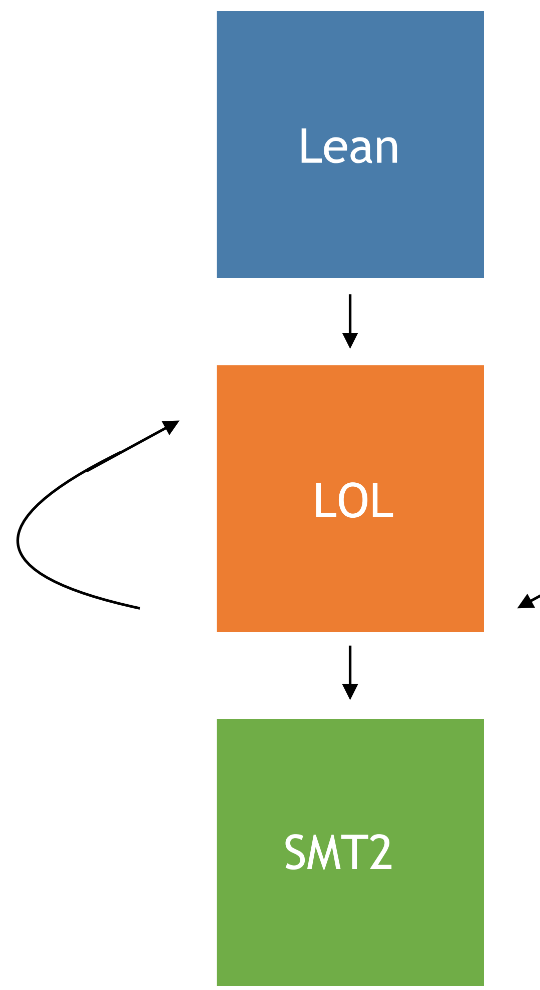
```
protected meta def transfer : tactic unit := do
  _root_.transfer.transfer [`relator.rel_forall_of_total, `dlist.rel_eq, `dlist.rel_empty,
    `dlist.rel_singleton, `dlist.rel_append, `dlist.rel_cons, `dlist.rel_concat]

example : ∀(a b c : dlist α), a ++ (b ++ c) = (a ++ b) ++ c :=
begin
  dlist.transfer,
  intros,
  simp
end
```

- We also use it to transfer results from nat to int.

Lean to SMT2

- Goal: translate a Lean local context, and goal into SMT2 query.
- Recognize fragment and translate to low-order logic (LOL).
- Logic supports some higher order features, is successively lowered to FOL, finally SMT2.




```
lemma n_gt_0
(a : nat) : a >= 0 :=
by z3
```

```
decl n : int {n >= 0}
assert (not (n >= 0))
```

```
(declare-const n Int)
(assert (>= n 0))
(assert (not (>= n 0))
```

```
mutual inductive type, term
with type : Type
| bool : type
| int : type
| var : string → type
| fn : list type → type → type
| refinement : type → (string → term) → type
with term : Type
| apply : string → list term → term
| true : term
| false : term
| var : string → term
| equals : term → term → term
| …
| forallq : string → type → term → term
```

```
meta structure context :=
(type_decl : rb_map string type)
(decls : rb_map string decl)
(assertions : list term)
```

```
meta def reflect_prop_formula' : expr → smt2_m lol.term
| `(¬ %%P) := lol.term.not <$> (reflect_prop_formula' P)
| `(%%P = %%Q) := lol.term.equals <$>
                  (reflect_prop_formula' P) <*>
                  (reflect_prop_formula' Q)
| `(%%P ∧ %%Q) := lol.term.and <$>
                  (reflect_prop_formula' P) <*>
                  (reflect_prop_formula' Q)
| `(%%P ∨ %%Q) := lol.term.or <$>
                  (reflect_prop_formula' P) <*>
                  (reflect_prop_formula' Q)
| `(%%P < %%Q) := reflect_ordering lol.term.lt P Q
| …
| `(true) := return $ lol.term.true
| `(false) := return $ lol.term.false
| e := …
```

Coinductive predicates

- Developed by Johannes Hölzl (approx. 800 lines of code)
- Uses impredicativity of Prop
- No kernel extension is needed

```
coinductive all_stream {α : Type u} (s : set α) : stream α → Prop
| step {} : ∀{a : α} {ω : stream α}, a ∈ s → all_stream ω → all_stream (a :: ω)
```

```
coinductive alt_stream : stream bool → Prop
| tt_step : ∀{ω : stream bool}, alt_stream (ff :: ω) → alt_stream (tt :: ff :: ω)
| ff_step : ∀{ω : stream bool}, alt_stream (tt :: ω) → alt_stream (ff :: tt :: ω)
```

simple expression language

```
inductive exp : Type
| Const (n : nat) : exp
| Plus (e1 e2 : exp) : exp
| Mult (e1 e2 : exp) : exp

def eeval : exp → nat
| (Const n)     := n
| (Plus e1 e2) := eeval e1 + eeval e2
| (Mult e1 e2) := eeval e1 * eeval e2

def times (k : nat) : exp → exp
| (Const n)     := Const (k * n)
| (Plus e1 e2) := Plus (times e1)
                       (times e2)
| (Mult e1 e2) := Mult (times e1) e2
```

```
def reassoc : exp → exp
| (Const n)     := (Const n)
| (Plus e1 e2) :=
  let e1' := reassoc e1 in
  let e2' := reassoc e2 in
  match e2' with
  | (Plus e21 e22) := Plus (Plus e1' e21) e22
  | _              := Plus e1' e2'
  end
| (Mult e1 e2) :=
  let e1' := reassoc e1 in
  let e2' := reassoc e2 in
  match e2' with
  | (Mult e21 e22) := Mult (Mult e1' e21) e22
  | _              := Mult e1' e2'
  end
```

nano crush

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

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

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

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

simple expression language

```
lemma eeval_times (k e) : eeval (times k e) = k * eeval e := by nano_crush
lemma reassoc_correct (e) : eeval (reassoc e) = eeval e := by nano_crush
```

Development support

- Profiler
  - Based on sampling
  - Useful for finding performance bottleneck in tactics
- Debugger based on VM monitor
  - User can write VM monitors in Lean
  - CLI debugger is implemented in Lean
  - IDE support is on the TODO list

vm monitor

```
meta structure vm_monitor (α : Type) := (init : α) (step : α → vm α)

meta def trace_step (p : tactic_state → name → bool) (sz : nat) : vm nat :=
do curr_sz ← call_stack_size, guard (sz ≠ curr_sz),
   ts      ← ts_from_current_frame,
   fn      ← curr_fn,
   when (p ts fn) $ do {
       put_str $ "tactic state at " ++ to_string fn ++ "\n",
       put_str $ to_string ts,
   }, return curr_sz

@[vm_monitor] meta def my_vm_monitor : vm_monitor nat :=
{ init := 0, step := trace_step (λ s fn, fn = ``nano_crush.search) }

set_option debugger true
lemma eeval_times (k e) : eeval (times k e) = k * eeval e := by nano_crush
```

Conclusion

- Users can create their on automation, extend and customize Lean
- Domain specific automation
- Internal data structures and procedures are exposed to users (e.g., congruence closure)
- White-box automation
- We are going to expose more
  - Structured trace messages
  - More powerful parser and pretty printing extensions
  - Code generator extensions
  - …