Fighting my Proof Anxiety with Lean
The Restricted Chase Indeed Yields Universal Models

Lukas Gerlach

Knowledge-Based Systems, TU Dresden

18.04.2024

What will we do today?

1. Define existential rules and the restricted chase in Lean (EZ but lengthy; bear with me)
2. Discuss key ingredients of the proof (Results you never thought you needed)
3. Lessons learned (What's cool; what's tedious?)

Existential Rules: $\text{Body}[\vec{x}, \vec{y}] \rightarrow \exists \vec{z} . \text{Head}[\vec{y}, \vec{z}]$

KnowledgeBaseBasics.lean

```
structure Rule where
  id : Nat
  body : FunctionFreeConjunction
  head : FunctionFreeConjunction

def FunctionFreeConjunction :=
  List FunctionFreeAtom

structure FunctionFreeAtom where
  predicate : Predicate
  terms : List VarOrConst
  deriving DecidableEq
```

Prelude

```
inductive List (α : Type u) where
| nil : List α
| cons (head : α) (tail : List α) : List α
```

TermBasics.lean

```
inductive VarOrConst where
| var (v : Variable) : VarOrConst
| const (c : Constant) : VarOrConst
deriving DecidableEq

structure Predicate where
  id : Nat
  deriving DecidableEq

structure Variable where
  id : Nat
  deriving DecidableEq

structure Constant where
  id : Nat
  deriving DecidableEq
```

Facts and Fact Sets: $\{P(\vec{t}), \ldots\}$

$$P(f_z^2(c, d))$$

KnowledgeBaseBasics.lean

```
def FactSet := Set Fact
def Database := Set FunctionFreeFact

structure Fact where
  predicate : Predicate
  terms : List GroundTerm
  deriving DecidableEq

structure FunctionFreeFact where
  predicate : Predicate
  terms : List Constant
```

TermBasics.lean

```
def GroundTerm :=
  FiniteTree SkolemFS Constant
deriving DecidableEq

structure SkolemFS where
  ruleId : Nat
  var : Variable
  deriving DecidableEq
```

Set.lean

```
def Set (α) := α -> Prop
```

Terms as Trees: $f_z^2(c, d)$

```
FiniteTree.inner ({ ruleId := 2, var := z : SkolemFS })
  (FiniteTreeList.cons (FiniteTree.leaf { id := 1 : Constant })
    FiniteTreeList.cons (FiniteTree.leaf { id := 2 : Constant })
      FiniteTreeList.nil)
```

FiniteTree.lean

```
mutual
  inductive FiniteTree (α : Type u) (β : Type v) where
    | leaf : β -> FiniteTree α β
    | inner : α -> FiniteTreeList α β -> FiniteTree α β

  inductive FiniteTreeList (α : Type u) (β : Type v) where
    | nil  : FiniteTreeList α β
    | cons : FiniteTree α β -> FiniteTreeList α β -> FiniteTreeList α β
end
```

Chase by Example

$$\rho = R(x, y) \to \exists z. R(y, z) \land R(z, y)$$

$$\rho_{\text{skolem}} = R(x, y) \to R(y, f_z^1(y)) \land R(f_z^1(y), y)$$

$$R(a, b)$$

The chase is a sequence of fact sets:

1. $\{R(a, b)\}$
2. $\{R(a, b), R(b, f_z^1(a)), R(f_z^1(a), b)\}$
3. $\{R(a, b), R(b, f_z^1(a)), R(f_z^1(a), b)\}$
4. ...

Formally, we consider triggers:

$$\langle \rho, [x/a, y/b] \rangle$$

The result of the trigger uses ρ_{skolem}.

Triggers: $\langle \rho, [x/c, ...] \rangle$

Trigger.lean

```
structure Trigger where
  rule : Rule
  subs : GroundSubstitution  -- def GroundSubstitution := Variable -> GroundTerm

def mapped_body (trg : Trigger) : List Fact := SubsTarget.apply trg.subs trg.rule.body

def loaded (trg : Trigger) (F : FactSet) : Prop := trg.mapped_body ⊆ F
def robsolete (trg : Trigger) (F : FactSet) : Prop :=
  ∃ s : GroundSubstitution,
    (∀ v, List.elem v (Rule.frontier trg.rule) → s v = trg.subs v) ∧
    ((s.apply_function_free_conj trg.rule.head).toSet ⊆ F)
def ractive (trg : Trigger) (F : FactSet) : Prop := trg.loaded F ∧ ¬ (trg.robsolete F)
```

Triggers: $\langle \rho, [x/c, ...] \rangle$

TermBasics.lean

```
def VarOrConst.skolemize (ruleId : Nat) (frontier : List Variable) (voc : VarOrConst)
  : SkolemTerm := match voc with
    | VarOrConst.var v => ite (List.elem v frontier) (SkolemTerm.var v)
      (SkolemTerm.func { ruleId := ruleId, var := v } frontier)
    | VarOrConst.const c => SkolemTerm.const c
```

Trigger.lean

```
[...]
def apply_to_var_or_const (trg : Trigger) : VarOrConst -> GroundTerm :=
  (trg.apply_to_skolemized_term ∘ trg.skolemize_var_or_const)
def apply_to_function_free_atom (trg : Trigger) (atom : FunctionFreeAtom) : Fact :=
  { predicate := atom.predicate, terms := atom.terms.map trg.apply_to_var_or_const }

def mapped_head (trg : Trigger) : List Fact :=
  trg.rule.head.map trg.apply_to_function_free_atom
def result (trg : Trigger) : FactSet := trg.mapped_head.toSet
```

Knowledge Bases and RTriggers

KnowledgeBaseBasics.lean

```
structure RuleSet where
  rules : Set Rule
  id_unique : ∀ r1 r2, r1 ∈ rules ∧ r2 ∈ rules ∧ r1.id = r2.id -> r1 = r2

structure KnowledgeBase where
  db : Database
  rules : RuleSet
```

Trigger.lean

```
def RTrigger (r : RuleSet) := { trg : Trigger // trg.rule ∈ r.rules}
```

Chase Sequences

ChaseSequence.lean

```
def InfiniteList (α : Type u) := Nat → α

structure ChaseSequence (kb : KnowledgeBase) where
  fact_sets : InfiniteList FactSet
  database_first : fact_sets 0 = kb.db
  triggers_exist : ∀ n : Nat, (
    ∃ trg : (RTrigger kb.rules),
      trg.val.ractive (fact_sets n) ∧ trg.val.result ∪ fact_sets n = fact_sets (n + 1)
  ) ∨ (
    ¬(∃ trg : (RTrigger kb.rules), trg.val.ractive (fact_sets n))
    ∧ fact_sets n = fact_sets (n + 1)
  )
  fairness : ∀ (trg : (RTrigger kb.rules)) (i : Nat),
    (trg.val.ractive (fact_sets i)) → ∃ j : Nat, j ≥ i ∧ (¬ trg.val.ractive (fact_sets j))

def ChaseSequence.result {kb : KnowledgeBase} (cs : ChaseSequence kb) : FactSet :=
  fun f => ∃ n : Nat, f ∈ cs.fact_sets n
```

Universal Models

Trigger.lean

```
namespace FactSet
  def modelsDb (fs : FactSet) (db : Database) : Prop := db.toFactSet ⊆ fs

  def modelsRule (fs : FactSet) (rule : Rule) : Prop :=
    ∀ trg : Trigger, (trg.rule = rule ∧ trg.loaded fs) -> trg.robsolete fs

  def modelsRules (fs : FactSet) (rules : RuleSet) : Prop :=
    ∀ r, r ∈ rules.rules -> fs.modelsRule r

  def modelsKb (fs : FactSet) (kb : KnowledgeBase) : Prop :=
    fs.modelsDb kb.db ∧ fs.modelsRules kb.rules

  def universallyModelsKb (fs : FactSet) (kb : KnowledgeBase) : Prop :=
    fs.modelsKb kb ∧
    (∀ m : FactSet, m.modelsKb kb -> ∃ (h : GroundTermMapping), isHomomorphism h fs m)
end FactSet
```

Fasten your seatbelts!

ChaseSequence.lean

```
theorem chaseResultUnivModelsKb (kb : KnowledgeBase) (cs : ChaseSequence kb) :
  cs.result.universallyModelsKb kb := by
  constructor; constructor
  -- DB in CS: FactSet.modelsDb (ChaseSequence.result cs) kb.db
  -- * every fact in kb.db is in 0th step of chase sequence by cs.database_first

  -- Rules modelled: FactSet.modelsRules (ChaseSequence.result cs) kb.rules
  -- * show: every loaded trigger is not ractive for cs.result
  -- * suppose it was ractive, then it is so for step i
  -- * by cs.fairness it is not ractive for some j>=i; thus not ractive for cs.result

  -- universality: ∀ (m : FactSet),
  --   FactSet.modelsKb m kb → ∃ h, isHomomorphism h (ChaseSequence.result cs) m
  -- * construct h inductively for each chase step (hardest part)
```

Non-Obvious Results Required

```
theorem subs_application_is_injective_for_freshly_introduced_terms {t : Variable} (trg :
Trigger) (t_not_in_frontier : ¬ trg.rule.frontier.elem t) : ∀ s, (trg.apply_to_var_or_const
(VarOrConst.var t) = trg.apply_to_var_or_const (VarOrConst.var s)) ->
trg.skolemize_var_or_const (VarOrConst.var t) = trg.skolemize_var_or_const (VarOrConst.var s)

theorem funcTermForExisVarInMultipleTriggersMeansTheyAreTheSame
  {rs : RuleSet} (trg1 trg2 : RTrigger rs) (var1 var2 : Variable)
  (var1_not_in_frontier : trg1.val.rule.frontier.elem var1 = false)
  (var2_not_in_frontier : trg2.val.rule.frontier.elem var2 = false) :
  (trg1.val.apply_to_var_or_const (VarOrConst.var var1)) = (trg2.val.apply_to_var_or_const
(VarOrConst.var var2)) -> trg1.val.rule = trg2.val.rule ∧ ∀ v, v ∈ trg1.val.rule.frontier.toSet
-> trg1.val.subs v = trg2.val.subs v

theorem funcTermForExisVarInChaseMeansTriggerIsUsed (kb : KnowledgeBase) (cs : ChaseSequence
kb) (trg : RTrigger kb.rules) (var : Variable) (i : Nat) : (trg.val.rule.frontier.elem var =
false) ∧ (∃ f: Fact, f ∈ cs.fact_sets i ∧ ((trg.val.apply_to_var_or_const (VarOrConst.var var))
∈ f.terms.toSet)) -> ∃ j, j ≤ i ∧ trg.val.result ∪ cs.fact_sets (j-1) = cs.fact_sets j
```

Lessons Learned

- Refactoring proofs is a pleasure (good luck on paper)
- Interactively writing proofs helps keeping focus

- Formalizing auxiliary results can be hard
- Standard library (e.g. for lists) could be better
- It takes time to get into thinking in Lean

Thanks!

```
-- Slides are built with Typst (of course)
theorem noBrainer : typst > latex := by trivial

-- TODO:
theorem sudokuTooEZ : P = NP := by sorry
```