Making Automatic Theorem Provers more Versatile

Simon Cruanes

Veridis, Inria Nancy
https://cedeela.fr/~simon/

August 2017
ATPs’ usefulness

ATPs are successfully applied:

- program verification (e.g., Boogie, Leon, Why3, F*…)
- automation in proof assistants (Sledgehammer, TLAPS, SMTCoq, …)
- synthesis (SyGuS)
- SAT/SMT in most symbolic methods
- …

(disclosure: here “ATP” means SMT or Superposition prover)
ATPs’ usefulness

ATPs are successfully applied:

- program verification (e.g., Boogie, Leon, Why3, F*…)
- automation in proof assistants (Sledgehammer, TLAPS, SMTCoq, …)
- synthesis (SyGuS)
- SAT/SMT in most symbolic methods
- …

(disclosure: here “ATP” means SMT or Superposition prover)

however! Problems often out of reach of ATPs…

…often because they live in a logic that is too expressive
ATPs’ Limitations

- SMT solvers have difficulties with quantifiers
  (incompleteness, sensitivity to input, mostly heuristics, etc.)
  - frame axioms in verification
  - many FO problems from Sledgehammer

- Superposition provers have troubles with theories
  - Arithmetic for most verification tasks
  - (co)datatypes for proof assistants POs

- both (usually) lack induction, HO, ...

- quantifiers + theories ⇒ even harder

- induction provers are usually bad on pure FO / theories
  (usually just Horn clauses + rewriting)
ATPs’ Limitations

- SMT solvers have difficulties with quantifiers (incompleteness, sensitivity to input, mostly heuristics, etc.)
  - frame axioms in verification
  - many FO problems from Sledgehammer

- Superposition provers have troubles with theories
  - Arithmetic for most verification tasks
  - (co)datatypes for proof assistants POs

- both (usually) lack induction, HO, ...

- quantifiers + theories ⇒ even harder

- induction provers are usually bad on pure FO / theories
  (usually just Horn clauses + rewriting)

- note: Progress on many aspects (CVC4+i, Vampire+z3, ...)
ATPs’ Limitations

- SMT solvers have difficulties with quantifiers (incompleteness, sensitivity to input, mostly heuristics, etc.)
  - frame axioms in verification
  - many FO problems from Sledgehammer
- Superposition provers have troubles with theories
  - Arithmetic for most verification tasks
  - (co)datatypes for proof assistants POs
- both (usually) lack induction, HO, ...
- quantifiers + theories \(\Rightarrow\) even harder
- induction provers are usually bad on pure FO / theories
  (usually just Horn clauses + rewriting)
- note: Progress on many aspects (CVC4+i, Vampire+z3, ...)

Current workarounds involve either encodings (e.g. Sledgehammer) or falling back to user (e.g. Why3 for inductive proofs)
Direction 1: Superposition $\oplus$ SMT

- SMT are excellent for ground reasoning with multiple theories
- Superposition provers are good for first-order reasoning
- Combining them: hot topic!
  - hierarchic superposition (Beagle)
    ($\triangleright$ no first-order theory reasoning)
  - AVATAR+T (Vampire)
    ($\triangleright$ completeness? explore combination with hierarchic sup)
  - using E as a SMT solver
    (will not do arithmetic)
  - DPLL($\Gamma + T$)
    ($\triangleright$ no competitive implementation yet)
Direction 1: Superposition + SMT

- SMT are excellent for ground reasoning with multiple theories
- Superposition provers are good for first-order reasoning
- Combining them: hot topic!
  - hierarchic superposition (Beagle)
    (no first-order theory reasoning)
  - AVATAR+T (Vampire)
    (completeness? explore combination with hierarchic sup)
  - using E as a SMT solver
    (will not do arithmetic)
  - DPLL(Γ + T)
    (no competitive implementation yet)

- Challenge: find a combination that
  - has good theoretical properties (at least completeness on FO, ground+T)
  - can be implemented efficiently
  - remains somehow elegant
With SMT, if a theory is not provided: **out of luck**

→ need to axiomatize
→ must learn black magic of triggers, etc.

**same holds for Superposition**
Direction 2 : User-defined Theories

- With SMT, if a theory is not provided: **out of luck**
  - → need to axiomatize
  - → must learn black magic of triggers, etc.

- same holds for Superposition

Possible solution: **Deduction Modulo Theory**

- Theory = set of *oriented* rewrite rules
- rules can apply to terms but also literals
- very useful for e.g.
  - ▶ set theory operators: \( x \in (A \cup B) \leadsto (x \in A \lor x \in B) \)
  - ▶ theory of (extensional) arrays

- not different from Superposition, except the *strategy* is different
- also useful for encodings and rec. functions (in Sledgehammer, . . .)

Simon Cruanes
Direction 3: Towards Higher-Order

Induction

1. “Sledgehammer is awesome” (users)
2. “lemma \( a + b = b + a \) by sledgehammer”
3. ...
4. \( \rightarrow \) No proof found

Higher-Order Reasoning

- proof assistants and functional languages are higher-order
- encodings are costly and inefficient
- Higher-Order ATPs are weak on first-order or propositional logic
- need first-order provers that are also decent at HO reasoning

(more details in next talk!)

Simon Cruanes

August 2017 6 / 7
### Induction

1. “Sledgehammer is awesome” (users)
2. “lemma $a + b = b + a$ by sledgehammer”
3. ...
4. → No proof found

provers need at least a basic notion of **induction**.
Direction 3: Towards Higher-Order

**Induction**

1. “Sledgehammer is awesome” (users)
2. “lemma \(a + b = b + a\) by sledgehammer”
3. ...
4. \rightarrow No proof found

provers need at least a basic notion of **induction**.

**Higher-Order Reasoning**

- proof assistants and functional languages are higher-order
- encodings are costly and inefficient
- Higher-Order ATPs are weak on first-order or propositional logic
  \rightarrow need first-order provers that are also decent at HO reasoning

(more details in next talk!)
we users need ATPs handling richer logics: quantifiers, higher-order, theories, induction, ...

3 directions (non exhaustive) which would improve this:

1. Combine Superposition and SMT
   → deals with FO + theories

2. Empower users with user-defined theories
   → possible solution: Deduction Modulo Theories (rewriting)

3. Basic support for induction and Higher-Order
   (I’ll let Jasmin talk about that)
we users need ATPs handling richer logics: quantifiers, higher-order, theories, induction, . . .

3 directions (non exhaustive) which would improve this:

1. Combine Superposition and SMT → deals with FO + theories
2. Empower users with user-defined theories → possible solution: Deduction Modulo Theories (rewriting)
3. Basic support for induction and Higher-Order (I’ll let Jasmin talk about that)

we have decent solutions to individual problems! challenge is how to combine in a single system (no portfolio!)
Questions

1. How to build a system for a combination of techniques (superposition+SMT+induction+...) with manageable complexity and correctness?

2. What theoretical framework would allow to describe such combinations in a simple(r) and general way?
val set : type \rightarrow type.

val[\text{infix } "\in"] mem : \pi a. a \rightarrow set a \rightarrow prop.
val[\text{infix } "\cup"] union : \pi a. set a \rightarrow set a \rightarrow set a.
val[\text{infix } "\subseteq"] subeq : \pi a. set a \rightarrow set a \rightarrow prop.

\text{rewrite} \forall a s1 s2 \times. \text{mem} a \times (\text{union} a s1 s2) \iff \text{mem} a \times s1 || \text{mem} a \times s2.

\text{rewrite} \forall a s1 s2. \text{sebeq} a s1 s2 \iff (\forall x. \text{mem} a \times s1 \implies \text{mem} a \times s2).

\text{rewrite} \forall a (s1 s2 : set a). s1 = s2 \iff (\text{subeq} s1 s2 && \text{subeq} s2 s1).

\text{goal} \\
\forall a (S1 S2 S3 S4 S5 S6 : set a).
\text{(union} S1 (\text{union} S2 (\text{union} S3 (\text{union} S4 (\text{union} S5 S6))))) \equiv \\
\text{(union} S6 (\text{union} S5 (\text{union} S4 (\text{union} S3 (\text{union} S2 S1))))).

Simon Cruanes
combine all the provers!
August 2017
solved in 0 steps

AVATAR does the splitting
entirely reduced to ∈-literals

bit-blasting for free!
Example

Classic theory of (extensional) arrays

val array : type -> type -> type.
val update : pi a b. array a b -> a -> b -> array a b.
val get : pi a b. array a b -> a -> b.

rewrite forall a b (arr:array a b) x1 x2 v.
  get (update arr x2 v) x1 = (if x1=x2 then v else get arr x1).

# extensionality by rewriting disequalities
rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <==> (forall x. get arr1 x = get arr2 x).
Example

Classic theory of (extensional) arrays

```agda
val array : type → type → type.
val update : pi a b. array a b → a → b → array a b.
val get : pi a b. array a b → a → b.

rewrite forall a b (arr:array a b) x1 x2 v.
  get (update arr x2 v) x1 = (if x1=x2 then v else get arr x1).

# extensionality by rewriting disequalities
rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <=> (forall x. get arr1 x = get arr2 x).
```

```agda
goal forall x arr. arr = update arr x (get arr x).
```
Example

Classic theory of (extensional) arrays

```ocaml
val array : type -> type -> type.
val update : pi a b. array a b -> a -> b -> array a b.
val get : pi a b. array a b -> a -> b.

rewrite forall a b (arr:array a b) x1 x2 v.
  get (update arr x2 v) x1 = (if x1=x2 then v else get arr x1).

# extensionality by rewriting disequalities
rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <=> (forall x. get arr1 x = get arr2 x).
```

**goal** forall x arr. arr = update arr x (get arr x).

**goal** forall x1 x2 arr. x1 != x2 && v1 != v2 =>
  update (update arr x1 v1) x2 v2 != update (update arr x2 v1) x1 v2.

Simon Cruanes