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, $F^\star$...)
- 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, $F^\star$...)
- 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 $\Rightarrow$ 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 $\Rightarrow$ 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, ...)

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 $\uplus$ 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(Γ + T)
    ($\triangleright$ no competitive implementation yet)

Direction 1 : Superposition $\uplus$ 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(Γ + T)
    ($\triangleright$ 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

Direction 2 : User-defined Theories

- With SMT, if a theory is not provided: **out of luck**
  - $\rightarrow$ need to axiomatize
  - $\rightarrow$ must learn black magic of triggers, etc.
- same holds for Superposition

- With SMT, if a theory is not provided: **out of luck**
  - $\rightarrow$ need to axiomatize
  - $\rightarrow$ 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: $x \in (A \cup B) \rightsquigarrow (x \in A \vee x \in B)$
  - theory of (extensional) arrays

$\rightarrow$ not different from Superposition, except the **strategy** is different

- also useful for encodings and rec. functions (in Sledgehammer, . . .)

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`

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.

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!)

Summary

- ~~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
     $\rightarrow$ deals with FO + theories
  2. Empower users with user-defined theories
     $\rightarrow$ possible solution: Deduction Modulo Theories (rewriting)
  3. Basic support for induction and Higher-Order
     (I'll let Jasmin talk about that)

Summary

- ~~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
     $\rightarrow$ deals with FO + theories
  2. Empower users with user-defined theories
     $\rightarrow$ 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 indivudual problems!
challenge is how to combine in a single system (no portfolio!)

Questions

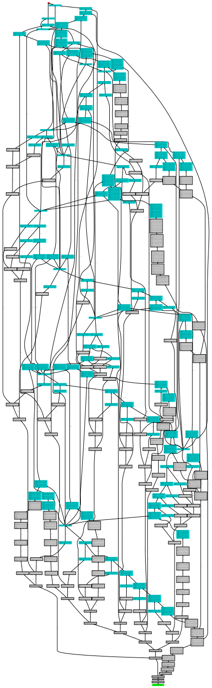
1. How to build a system for a combination of techniques (superposition+SMT+induction+. . .) with manageable complexity and correctness?
2. What theoretical framework would allow to describe such combinations in a simple(r) and general way?

Deduction Modulo Example : Set Theory

```
val set : type -> type.

val[infix "∈"] mem : pi a. a -> set a -> prop.
val[infix "∪"] union : pi a. set a -> set a -> set a.
val[infix "⊆"] subeq : pi a. set a -> set a -> prop.

rewrite forall a s1 s2 x.  mem a x (union a s1 s2) <=> mem a x s1 || mem a x
s2.

rewrite forall a s1 s2.  subeq a s1 s2 <=> (forall x. mem a x s1 => mem a x
s2).

rewrite forall a (s1 s2 : set a).  s1 = s2 <=> (subeq s1 s2 && subeq s2 s1).

goal
  forall a (S1 S2 S3 S4 S5 S6 : set a).
    (union S1 (union S2 (union S3 (union S4 (union S5 S6))))) =
    (union S6 (union S5 (union S4 (union S3 (union S2 S1))))).
```

- solved in 0 steps
- entirely reduced to $\in$-literals
- AVATAR does the splitting

$\rightarrow$ bit-blasting for free!



Lightweight Theories Example : Arrays

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).
```

Lightweight Theories Example : Arrays

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).
```

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

Lightweight Theories Example : Arrays

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).
```

```
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.
```