Integration of SMT to Coq

Seminar: Satisfiability Checking

David Kräutmann

RWTH Aachen

November 12, 2019

Outline

Introduction

Interactive theorem provers used in

- Formalized proofs
  - Four color theorem
  - Odd order theorem

  $\Rightarrow$ High amount of cases, paper proofs unfeasible

Introduction

Interactive theorem provers used in

- Formalized proofs
  - Four color theorem
  - Odd order theorem
  - $\Rightarrow$ High amount of cases, paper proofs unfeasible
- Verified computation
  - CompCert
  - Compiler transformations
  - High-assurance programs
  - $\Rightarrow$ Many trivial cases, few interesting ones

Introduction

Interactive theorem provers used in

- Formalized proofs
  - Four color theorem
  - Odd order theorem
  - $\Rightarrow$ High amount of cases, paper proofs unfeasible
- Verified computation
  - CompCert
  - Compiler transformations
  - High-assurance programs
  - $\Rightarrow$ Many trivial cases, few interesting ones
- Foundations of mathematics
  - Univalent Mathematics
  - $\Rightarrow$ Formalised in Coq from the start

Automation

- Formal proofs contain many trivial or similar cases
- Informal proofs handwave those $\Rightarrow$ Potential for errors
- Proof assistants require automation
  - Built-in tactics: `auto`, `omega`
  - External solvers: SMTCoq, Ergo-Coq
  - User-written custom tactics

About Coq

- Proof assistant based on type theory
- Similar to functional programming languages
- Both a pure functional language and proof development system

Programming with Coq

```
1  Section list.
2    Variable A : Type.
3
4    Inductive list : Type :=
5    | nil : list
6    | cons : A -> list -> list.
7
8    Fixpoint concat (xs : list) (ys : list) :=
9      match xs with
10       | nil => ys
11       | cons x xs => cons x (concat xs ys)
12     end.
13 End list.
```

Proving properties

```
1  Theorem concat_assoc : forall xs ys zs,
2      concat (concat xs ys) zs = concat xs (concat ys zs).
3    induction xs.
4    intros. simpl. reflexivity.
5    intros. simpl. rewrite IHxs. reflexivity.
6  Qed.
```

Inductive types

- Generalisation of algebraic data types
- Regular type `Definition`: just an alias
- `Inductive` type: automatic induction principle

Natural numbers

```
1  Inductive nat : Set :=
2  | O : nat
3  | S : nat -> nat.
```

Definitions, fixpoints, theorems

```
1  Definition zerob (n : nat) :=
2    match n with
3    | O => true
4    | S _ => false
5    end.
```

Definitions, fixpoints, theorems

```
1  Fixpoint plus (n : nat) (m : nat)  :=
2    match n with
3    | O => m
4    | S n' => S (plus n' m)
5    end.
```

```
plus is defined
plus is recursively defined (decreasing on 1st argument)
```

Definitions, fixpoints, theorems

```
1 Theorem n_plus_0 : forall n : nat, plus n 0 = n.
```

```
1 subgoal

  ============================
  forall n : nat, plus n 0 = n
```

Definitions, fixpoints, theorems

```
1 Theorem n_plus_0 : forall n : nat, plus n 0 = n.
2   induction n.
```

```
2 subgoals

  ============================
  plus 0 0 = 0

subgoal 2 is:
 plus (S n) 0 = S n
```

Definitions, fixpoints, theorems

```
1 Theorem n_plus_O : forall n : nat, plus n O = n.
2   induction n.
3   reflexivity.
```

```
1 subgoal

  n : nat
  IHn : plus n 0 = n
  ============================
  plus (S n) 0 = S n
```

Definitions, fixpoints, theorems

```
1 Theorem n_plus_O : forall n : nat, plus n O = n.
2   induction n.
3   reflexivity.
4   simpl.
```

```
1 subgoal

  n : nat
  IHn : plus n 0 = n
  ============================
  S (plus n 0) = S n
```

Definitions, fixpoints, theorems

```
1 Theorem n_plus_O : forall n : nat, plus n O = n.
2   induction n.
3   reflexivity.
4   simpl.
5   rewrite IHn.
```

```
1 subgoal

  n : nat
  IHn : plus n 0 = n
  ============================
  S n = S n
```

Definitions, fixpoints, theorems

```
1  Theorem n_plus_O : forall n : nat, plus n O = n.
2    induction n.
3    reflexivity.
4    simpl.
5    rewrite IHn.
6    reflexivity.
7  Qed.
```

```
n_plus_O is defined
```

Polymorphic types

```
1  Inductive list (A : Type) : Type :=
2    | nil : list A
3    | cons : A -> list A -> list A.
```

Polymorphic types

```
1  Section list.
2    Variable A : Type.
3
4    Inductive list : Type :=
5    | nil : list
6    | cons : A -> list -> list.
7  End list.
```

Polymorphic types

```
1  Section list.
2    Variable A : Type.
3
4    Inductive list : Type :=
5    | nil : list
6    | cons : A -> list -> list.
7
8    Fixpoint concat (xs : list) (ys : list) :=
9      match xs with
10       | nil => ys
11       | cons x xs => cons x (concat xs ys)
12     end.
13 End list.
```

Proof automation

```
1  Theorem concat_assoc : forall xs ys zs,
2      concat (concat xs ys) zs = concat xs (concat ys zs).
3    induction xs.
4    intros. simpl. reflexivity.
5    intros. simpl. rewrite IHxs. reflexivity.
6  Qed.
```

Proof automation

```
1  Theorem concat_assoc : forall xs ys zs,
2      concat (concat xs ys) zs = concat xs (concat ys zs).
3    induction xs; intros; simpl.
4    reflexivity.
5    rewrite IHxs. reflexivity.
6  Qed.
```

Proof automation

```
1  Require Import CpdtTactics.
2
3  Theorem concat_assoc : forall xs ys zs,
4      concat (concat xs ys) zs = concat xs (concat ys zs).
5    induction xs; crush.
6  Qed.
```

Predicates

- Encoding properties
- Usually defined via `Inductive`
- Each condition is a constructor
- Condition fulfilled when all arguments are supplied

Example predicate

```
1  Inductive isEven : nat -> Prop :=
2  | Even_0 : isEven 0
3  | Even_SS : forall n, isEven n -> isEven (S (S n)).
```

Example predicate theorem

```
1 Theorem even_plus : forall n m,
2     isEven n -> isEven m -> isEven (n + m).
```

```
1 subgoal

  ============================
  forall n m : nat, isEven n -> isEven m -> isEven (n + m)
```

Example predicate theorem

```
1 Theorem even_plus : forall n m,
2     isEven n -> isEven m -> isEven (n + m).
3   intros n m nEven mEven.
```

```
1 subgoal

  n, m : nat
  nEven : isEven n
  mEven : isEven m
  ============================
  isEven (n + m)
```

Example predicate theorem

```
1  Theorem even_plus : forall n m,
2      isEven n -> isEven m -> isEven (n + m).
3    intros n m nEven mEven.
4    induction nEven.
```

```
2 subgoals

  m : nat
  mEven : isEven m
  ============================
  isEven (0 + m)

subgoal 2 is:
 isEven (S (S n) + m)
```

Example predicate theorem

```
1  Theorem even_plus : forall n m,
2      isEven n -> isEven m -> isEven (n + m).
3    intros n m nEven mEven.
4    induction nEven. apply mEven.
```

```
1 subgoal

  m, n : nat
  nEven : isEven n
  mEven : isEven m
  IHnEven : isEven (n + m)
  ============================
  isEven (S (S n) + m)
```

Example predicate theorem

```
1  Theorem even_plus : forall n m,
2      isEven n -> isEven m -> isEven (n + m).
3    intros n m nEven mEven.
4    induction nEven. apply mEven.
5    simpl.
```

```
1 subgoal

  m, n : nat
  nEven : isEven n
  mEven : isEven m
  IHnEven : isEven (n + m)
  ============================
  isEven (S (S (n + m)))
```

Example predicate theorem

```
1  Theorem even_plus : forall n m,
2      isEven n -> isEven m -> isEven (n + m).
3    intros n m nEven mEven.
4    induction nEven. apply mEven.
5    simpl. apply Even_SS.
```

```
1 subgoal

  m, n : nat
  nEven : isEven n
  mEven : isEven m
  IHnEven : isEven (n + m)
  ============================
  isEven (n + m)
```

Example predicate theorem

```
1  Theorem even_plus : forall n m,
2      isEven n -> isEven m -> isEven (n + m).
3    intros n m nEven mEven.
4    induction nEven. apply mEven.
5    simpl. apply Even_SS.
6    apply IHnEven.
7  Qed.
```

```
even_plus is defined
```

Connectives

- Connectives like $\neg$ or $\wedge$ are not built-in.
- Only built-in types: dependent product (forall) and function type

Connective definitions

```
1  Inductive False : Prop := .
```

```
1  Definition not P := P -> False.
```

```
1  Inductive and (A : Prop) (B : Prop) : Prop :=
2  | conj : A -> B -> and A B.
```

```
1  Inductive or (A : Prop) (B : Prop) : Prop :=
2  | or_introl : A -> or A B
3  | or_intror : B -> or A B.
```

Dependent types

- Dependent types are everywhere in `Prop`
  - `Inductive`: `isEven`

Dependent types

```
1  Inductive isEven : nat -> Prop :=
2  | Even_0 : isEven 0
3  | Even_SS : forall n, isEven n -> isEven (S (S n)).
```

Dependent types

- Dependent types are everywhere in `Prop`
  - `Inductive`: `isEven`
  - `Theorem`: everywhere

Dependent types

```
1  Theorem n_plus_O : forall n : nat, plus n O = n.
2    induction n.
3    reflexivity.
4    simpl.
5    rewrite IHn.
6    reflexivity.
7  Qed.
```

Dependent types

- Dependent types are everywhere in `Prop`
  - `Inductive`: `isEven`
  - `Theorem`: everywhere
- But also useful in `Set`
  - Length-indexed lists
  - Verified red-black trees

Length-indexed lists

```
1  Section ilist.
2    Variable A : Type.
3
4    Inductive ilist : nat -> Type :=
5    | inil : ilist 0
6    | icons : forall {n}, A -> ilist n -> ilist (S n).
7
8    Fixpoint concat {n} {m} (xs : ilist n) (ys : ilist m)
9        : ilist (n + m) :=
10     match xs with
11       | inil => ys
12       | icons x xs => icons x (concat xs ys)
13     end.
```

Length-indexed lists

```
1  Fixpoint inject (xs : list A) : ilist (length xs) :=
2    match xs with
3      | nil => inil
4      | cons x xs => icons x (inject xs)
5    end.
```

Length-indexed lists

```
1    Definition hd {n} (xs : ilist (S n)) : A :=
2      match xs with
3        (* Coq knows that inil is impossible *)
4        | icons x _ => x
5      end.
6  End ilist.
```

SAT/SMT integration

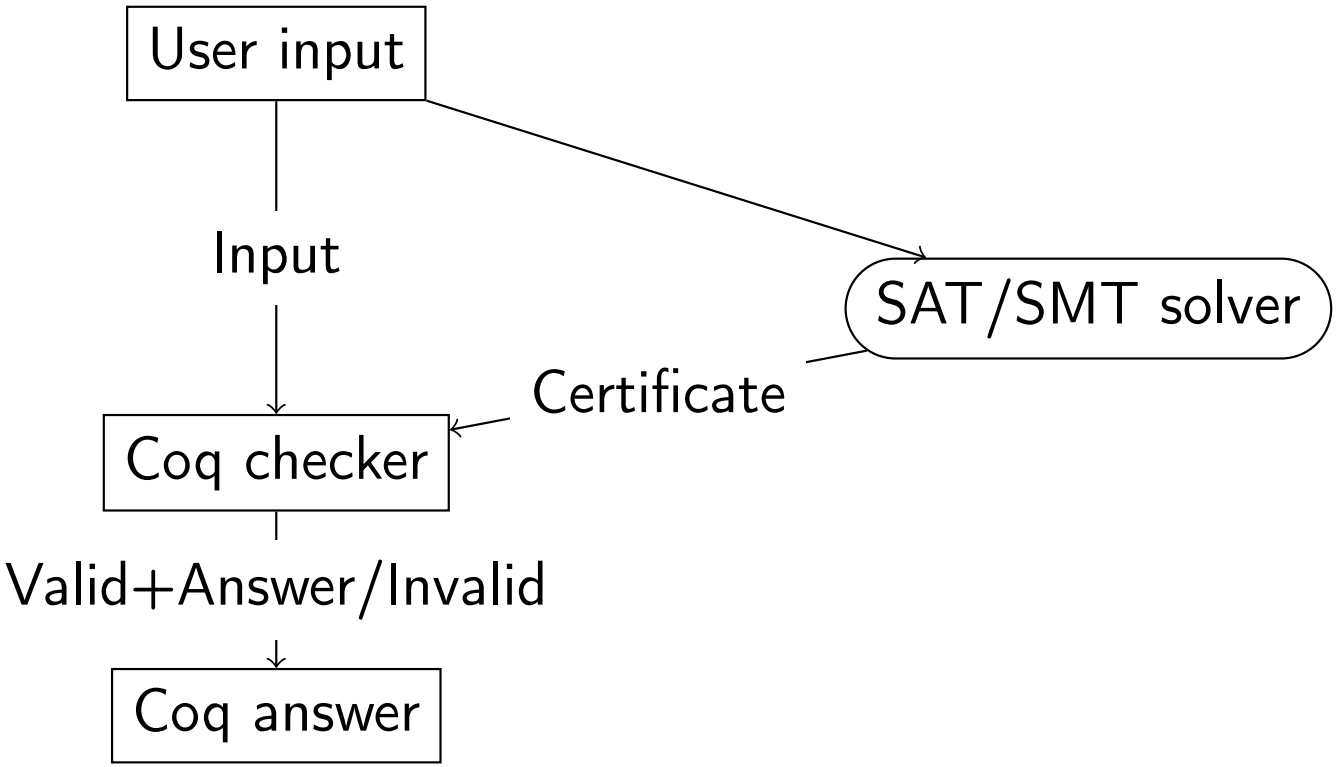
- Formal proofs need automation for practicality
- Integrating SAT/SMT allows to delegate tedious proofs

```
1 Theorem plus_example : forall n m : nat,
2     m < n -> n + m < 2 * n.
3   intros. omega.
4 Qed.
```

Two main approaches

- Skeptical approach
  - Transform goal into appropriate input
  - Call external solver to generate certificate
  - Verify certificate correctness
- Autarkic approach
  - Solver embedded inside theorem prover
  - Proven correct and total

Overview skeptical approach



Example: SAT solver

- User input: arbitrary CNF boolean formula
- SAT solver: any external solver like ZChaff
- Certificate: Resolution chain or variable assignment
- Checker verifies that certificate is correct

  Example: $x \wedge \neg x$ is UNSAT.

  Valid when called with resolution chain $\dfrac{x \quad \neg x}{\square}$

  Invalid when called with assignment $x = \top$

Correctness

- Checker is verified correct and total

$\Rightarrow$ Checker returns "valid" $\Longrightarrow$ formula has solution

- External solver can fail
  - Existing solution not found
  - Invalid certificate
  - Non-termination

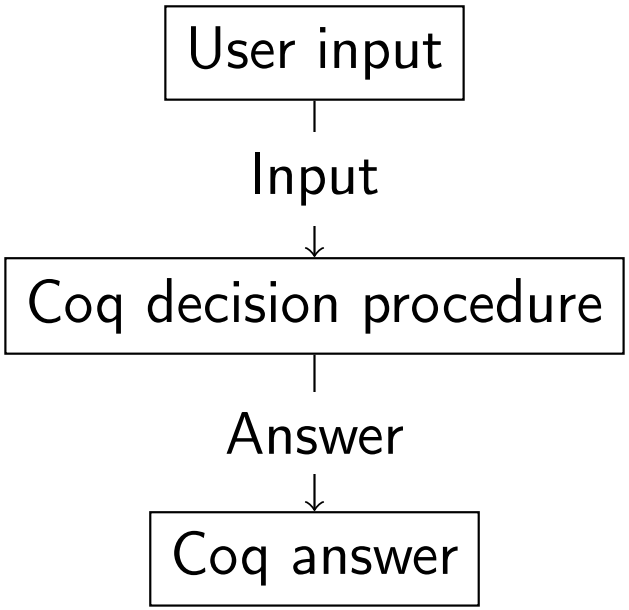
SMTCoq

- Skeptical SAT/SMT integration: SMTCoq
- Provides tactics for
  - SAT over `bool`
  - Uninterpreted functions
  - Linear integer arithmetic
- Not as comfortable to use as tactics over `Prop`
- Successfully used in sub-proofs of the odd-order theorem

Conclusion

- Benefits
  - Checker can be used by other programs
  - Better performance than autarkic methods
- Difficulties
  - Efficient transformation from Coq to CNF
  - Issues with constructive logic
  - Performant checker implementation

Overview autarkic approach



Comparison to skeptical approach

- Decision procedure (e.g. SAT solver) verified correct and total.

$\Rightarrow$ Stronger correctness guarantess

- Drawback: *Everything* has to be proven correct
  - Termination
  - Correctness of efficiency-improving transformations

$\Rightarrow$ Difficult! e.g. 9 line correctness proof on paper $\rightarrow$ over 700 line proof script

Why are formal proofs difficult?

- Informal proofs lack details:
  - Existence proofs are omitted
  - Inductive reasoning often implicit: $(K_1, ...K_i, ...K_n)$ so that K_i ...

$\Rightarrow$ Potential source of errors

$\Rightarrow$ But also appreciated: too many details make proofs cumbersome

- Induction difficult for current automated provers

Autarkic Coq tactics

- Simple DPLL procedure: `unsat`
- Quantifier-free Presbuger arithmetic: `omega`
- Hybrid: Ergo-Coq
  - Core of Alt-Ergo proven correct in Coq
  - Integrated back into Coq

Comparison between Ergo-Coq and SMTCoq

- Skeptical tactic: SMTCoq with `zchaff`, `verit`
- Autarkic tactic: Ergo-Coq with `dpll`, `cc`

SAT	`dpll`	`zchaff`	EUF	`cc`	`verit`
deb_{700}	111.5	0.8	D_5	2.3	0.3
deb_{800}	147.9	1.0	D_8	24.9	1.1
deb_{900}	201.6	1.2	D_{10}	118.7	2.2
deb_{1000}	260.4	1.5	D_{15}	-	45.7

Table: Benchmarks from [1]

Conclusion

- Writing formal proofs is (still) difficult
- Skeptical approach order of magnitudes faster, but less integrated
- Autarkic approach similar in usage to other Coq tactics
- Induction a major difficulty in automation

Related works

- Coq books:
  - Coq'Art [2]
  - CPDT [3]
- Interactive Coq course: Software Foundations [4]
- Similar languages: Agda, Idris, Isabelle/HOL, ...

References I

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq Through Proof Witnesses. In *Proceedings of the First International Conference on Certified Programs and Proofs*, CPP'11, pages 135–150, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Y. Bertot, P. Castéran, G. i. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Données complémentaires http://coq.inria.fr.

[3] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

References II

[4] B. C. Pierce et al. Software Foundations. Available at `https://www.cis.upenn.edu/~bcpierce/sf/current/index.html`.

Logic and Type Theory

Higher-order intuitionistic type theory

Logic and Type Theory

Higher-order

- First-order: quantification over elements
  $\forall x : \mathbb{N},\ x + 0 = x$
- Second-order: quantification over sets of elements
  $\forall P \forall x,\ Px \vee \neg Px$
- Higher-order: quantification over sets of sets

Logic and Type Theory

Higher-order intuitionistic

- Proofs are constructive
- No excluded middle ($x \vee \neg x$) or equivalent
- Classical logic: theorem is true or false
- Intuitionistic logic: theorem is proven or unproven

$\Rightarrow$ Benefit: Proofs are programs and programs are proofs

$\Rightarrow$ Consequence: Not all classical theorems are provable

Logic and Type Theory

Higher-order intuitionistic type theory

- Calculus of Inductive Constructions
- No underlying axioms, only inference rules
- Dependent types and inductive types
- No strict separation between proofs and programs (cf. set theory)

Example classical theorem

De Morgan's laws

$$\neg(P \wedge Q) \implies \neg P \vee \neg Q \tag{1}$$
$$\neg(P \vee Q) \implies \neg P \wedge \neg Q \tag{2}$$
$$\neg P \vee \neg Q \implies \neg(P \wedge Q) \tag{3}$$
$$\neg P \wedge \neg Q \implies \neg(P \vee Q) \tag{4}$$

(1) is unprovable in intuitionistic logic — requires law of excluded middle.

Curry-Howard isomorphism

- Relationship between logic and computation

Logic	Programming
Proposition	Type
Proof	Program
Provability	Type inhabitation problem

Curry-Howard isomorphism

- Relationship between logic and computation

Logic	Programming
Implication ($A \implies B$)	Function type (`A -> B`)
Conjunction ($A \wedge B$)	Product type (`A * B`)
Disjunction ($A \vee B$)	Sum type (`A + B`)

Booleans and totality

```
1  Inductive bool : Set :=
2    | true
3    | false.
```

- Note: all Coq functions are total

$\Rightarrow$ All functions returning `bool` are decidable

- Propositions like `True` or `False` can be undecidable
  Example: equality between functions `f g : nat -> nat`
- Still useful as hypotheses!