Integration of SMT to Coq

Seminar: Satisfiability Checking

David Kraeutmann
Supervision: Florian Frohn

SS 2016

Abstract

Coq is an interactive theorem prover used for many computerized proofs, such as the proof of the four-color theorem [10]. We give a short overview over Coq and discuss methods of integrating it with SAT/SMT solvers. These integrations enhance Coq's existing automation tactics and improve the reliability of automated provers by using a verified program to check their results.

1 Introduction

The usage of interactive theorem provers for both complex proofs [10, 12] and for designing a foundation for mathematics [17] has been steadily increasing over the years. However, unlike in informal paper proofs you cannot simply handwave the tedious "trivial cases". This means that a large amount of automation is needed in order for proof complexity to remain manageable.

Coq already supports a decent amount of decision and semi-decision procedures [6] such as `auto` (basic proof tree search), `omega` (solver for Presburger arithmetic) and many others. Nevertheless, advances in SMT solvers have not yet carried over into interactive theorem proving, both for performance reasons and difficulties in proving the formal correctness of algorithms used [15].

In Section 2 we provide an introduction to Coq and the underlying type theory, the Calculus of Inductive Constructions. Section 3 then details various approaches to using SAT/SMT solvers in conjunction with Coq.

2 The Coq proof assistant

Coq [7, 6] is a proof assistant based on the Calculus of Inductive Constructions (CIC) [8, 9], a type theory with dependent and inductive types. It is both a pure functional programming language and a proof development system.

Coq's underlying logic can be described as a higher-order intuitionistic predicate logic. The major difference to classical logic is the absence of the law of excluded middle $p \vee \neg p$. Formulae in intuitionistic logic are not 'true' or 'false' but instead are proven or disproven. The primary benefit of that is that under the Curry-Howard isomorphism, programs written in the Calculus of Inductive Constructions have a direct syntactic correspondence to proofs in higher-order intuitionistic predicate logic, with the proven theorem being the type of the program. Thus when proving something in Coq, it's possible to *run* the proof and compute the resulting object — for example, a proof of

`forall n m : nat, exists q : nat, exists r : nat, n = q * m + r` can be run in order to compute q and r.

As a consequence, not all theorems that are true in classical logic are provable in intuitionistic logic. For example, consider the De Morgan's laws

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

Out of these, (1) is unprovable in intuitionistic logic, as it requires the law of excluded middle to prove.

2.1 Programs are proofs

Consider the trivial proposition `True`, consisting of an inductive type with one nullary constructor and the unprovable proposition `False` with no constructors: The proposition

```
Inductive True : Prop :=
  | I.
```

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

that `True` implies itself has type `True -> True` and is proven by `fun x : True => x`. Similarly *ex falso quodlibet* can be expressed with the following theorem and (tactic-based) proof:

```
Theorem ex_falso : forall P : Prop, False -> P.
  intros P H.
  case H.
Qed.
```

This introduces two new hypotheses, `P : Prop` and `H : False` and then does case analysis on H. Since False has no constructors, this completes the proof. We'll come back to the concept of tactics later.

Checking the generated program using `Print ex_falso.` we see:

```
ex_falso = fun (P : Prop) (H : False)
  => match H with (* False has no constructors *) end
```

2.2 Inductive types

Coq's inductive types are a generalisation of algebraic data types that exist in most functional languages. The singleton type `unit` is defined by

```
Inductive unit : Set :=
  | tt.
```

The difference between inductive type definitions and other kinds of definitions such as `Definition` is that inductive types automatically admit an induction principle, as seen in this theorem and proof:

```
Theorem unit_is_singleton : forall x : unit, x = tt.
  induction x. (* goal is now tt = tt *)
  reflexivity. (* which holds because of reflexivity *)
Qed.
```

In this case, the induction principle is

```
unit_ind : forall P : unit -> Prop, P tt -> forall u : unit, P u
```

Astute readers notice that `unit` and `True` are very similar – the only difference being that one is in `Set` while the other is in `Prop`. In general, `Set` is the type of types used in programming and `Prop` is the type of propositions. This convention is important for several reasons. First, it allows you to *extract* a program living in `Set` to an easier to optimize language such as Haskell while ignoring the (runtime-irrelevant) proofs in `Prop`. Secondly, `Prop` is impredicative, meaning that quantifying over `Prop` also yields `Prop`:

```
(forall T : Prop, T) : Prop
(forall P Q : Prop, P \/ Q -> Q \/ P) : Prop
(forall T : Set, T) : Type (* not Set! *)
```

This is mostly for consistency reasons when combined with the law of excluded middle — an impredicative `Set` causes inconsistency when axioms of choice are used in combination with e.g. excluded-middle [6].

Similar to other functional languages we can define the type of booleans `bool` as

```
Inductive bool : Set :=
  | true
  | false.
```

Note that `true` and `false` are completely different from inhabited and uninhabited propositions. All functions have to be total, which means that functions returning `bool` have to be decidable. Propositions, on the other hand, can be undecidable — a proposition is only decidable when `P \/ ~P` holds (without assuming excluded-middle). In constructive logic, that statement means that we can construct a proof object for `P` or `P -> False`. It's clear that that doesn't hold for every `Prop`; for instance, equality between natural numbers is decidable (either `n = m` or `n <> m`), but equality between functions from natural numbers to themselves is not: asserting the equality of two arbitrary functions `f g : nat -> nat` requires one to check an infinite amount of values. However, being able to use such a theorem is still useful, since you can use it as a hypothesis when proving other theorems.

All the examples above don't actually require the induction principle – since there is no recursion, simple case analysis is sufficient. The natural numbers are a type where induction is actually required:

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

We can pattern match on it with `match`, write terminating recursive functions with `Fixpoint` and define and prove theorems about it with `Theorem`

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

```
Fixpoint plus (n : nat) (m : nat)  :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

```
Theorem n_plus_O : forall n : nat, plus n O = n.
  induction n.
  reflexivity. (* plus 0 0 = 0 is trivial *)
  simpl. (* simplify (plus (S n) 0 = S n) to (S (plus n 0) = S n) *)
  rewrite IHn. (* use the induction hypothesis IHn : plus n 0 = n *)
  reflexivity. (* S n = S n *)
Qed.
```

We can also define polymorphic types. Here we use `Section` to avoid repeating arguments — the following two snippets are equivalent.

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

```
Section list.
  Variable A : Type.

  Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
End list.
```

Note how we quantify over `Type`! This way `list` works on both `Prop` and `Set`, since `Type` is a supertype of both.

Functions over list are similar to their Haskell equivalents (the following examples are inside `Section list`):

```
Fixpoint length (xs : list) :=
  match xs with
    | nil => O
    | cons _ xs => S (length xs)
  end.
```

```
Fixpoint concat (xs : list) (ys : list) :=
  match xs with
    | nil => ys
    | cons x xs => cons x (concat xs ys)
  end.
```

Coq automatically verified that `length` and `concat` are terminating. Now we can prove properties over lists such as associativity of `concat`:

```
Theorem concat_assoc : forall xs ys zs,
    concat (concat xs ys) zs = concat xs (concat ys zs).
  induction xs.
  intros. simpl. reflexivity.
  intros. simpl. rewrite IHxs. reflexivity.
Qed.
```

There is some duplication here. The only difference between the cases is the `rewrite IHxs` step where the induction hypothesis is applied. We can automate this a little with the ; operator, which applies the given tactic to all generated subgoals. Using that, we can write the proof as

```
Theorem concat_assoc : forall xs ys zs,
    concat (concat xs ys) zs = concat xs (concat ys zs).
  induction xs; intros; simpl.
  reflexivity.
  rewrite IHxs. reflexivity.
Qed.
```

Certified Programming with Dependent Types (CPDT) [6] features the custom tactic `crush`, which tries a variety of tactics to solve the goal. Using it, the proof is as simple as `induction xs; crush`. However, there is not enough space to explain its internals properly. Thus we will avoid using it, despite much shorter resulting proofs.

2.3 Predicates

Just like types, predicates are defined via `Inductive`. Informally, inductive predicates are used whenever we want x to have property P if and exactly if a condition from a set of conditions is fulfilled. Each condition is a constructor and it's fulfilled exactly when all arguments can be supplied. For instance, evenness of natural numbers can be encoded by

```
Inductive isEven : nat -> Prop :=
| Even_O : isEven O
| Even_SS : forall n, isEven n -> isEven (S (S n)).
```

and of course we can prove properties of even numbers, such as invariance under addition.

```
Theorem even_plus : forall n m,
    isEven n -> isEven m -> isEven (n + m).
  intros n m nEven mEven.
  induction nEven. simpl. apply mEven.
  simpl. apply Even_SS. apply IHnEven.
Qed.
```

First we introduce the hypotheses `n m : nat, nEven : isEven n` and `mEven : isEven m`. Then we do induction on `nEven`, generating two subgoals. The first, `isEven (0 + m)`, is solved by simplifying to `isEven m` and then applying the hypothesis `mEven`.
In the second goal, `isEven (S (S n) + m)`, we additionally have the induction hypotheses `IHnEven : isEven (n + m)`. Again we simplify the goal to `isEven (S (S (n + m)))`. This allows us to apply `Even_SS` as the goal matches the right side of that constructor. This leaves us with `isEven (n + m)`, which is proven by `apply IHnEven`.

2.3.1 Connectives

Connectives from propositional logic are also types — `not` is a simple type definition without any induction principles, thus `Definition` is sufficient. Everything else is inductive, as we need to be able to pattern match or do induction on those.

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

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

Inductive or (A : Prop) (B : Prop) : Prop :=
| or_introl : A -> or A B
| or_intror : B -> or A B.

Inductive ex {A : Type} (P : A -> Prop) : Prop :=
| ex_intro : forall x, P x -> ex P.
```

`not P` can be written as `~P` and `and A B` has the associated operator `A /\ B`. Similarly, `or A B` and `ex P` have the notations `A \/ B` and `exists x, P x`.

Some more theorems to demonstrate various tactics:

```
Theorem doublethink : ~(2 + 2 = 5).
  discriminate 1.
Qed.
```

`discriminiate` proves any goal when a hypothesis (in this case we use the first argument of `not`, indicated by the `1`) is built by different constructors.

```
Theorem and_comm : forall P Q,
    P /\ Q -> Q /\ P.
  induction 1.
  split; assumption.
Qed.
```

`split` is generally used with `and` and applies the first constructor to the goal. Since `conj : A -> B -> and A B` this generates a goal `A` and a goal `B`. `assumption` looks for a previously introduced hypothesis whose type is equal to the goal and applies it if found, solving the goal.

2.4 Dependent types

We've already seen usage of dependent types in the definition of `isEven` – we take a `nat` as a type argument, leading to `Even_0 : isEven 0`. Of course, we can also use dependent types outside of predicates. The standard example is the length-indexed list, also called a vector.

```
Section ilist.
  Variable A : Type.
  Inductive ilist : nat -> Type :=
  | inil : ilist 0
  | icons : forall {n}, A -> ilist n -> ilist (S n).

  Fixpoint concat {n} {m} (xs : ilist n) (ys : ilist m) : ilist (n + m) :=
    match xs with
      | inil => ys
      | icons x xs => icons x (concat xs ys)
    end.
```

Of course we can convert a regular `list` to an indexed `ilist`.

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

The benefit is obvious – we can have a safe head function without Maybe:

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

A possible use case for length-indexed lists exists in vector math. For example, the addition of two elements of a Cartesian vector space is only possible when the dimensions match. A Coq function would enforce this by having a type signature similar

to `add : forall n, ilist n real -> ilist n real -> ilist n real`. In other languages, this would often require one to either create a hard-coded set of types like `V2`, `V3`, ... —this approach is used in the Haskell package `linear`— or rely on run-time errors when dimensions mismatch.

While these examples are rather straightforward, working with dependent types can often stress the limits of Coq's type inference heuristics, and sometimes encoding properties directly into a data structure can require intermediary types to temporarily suspend invariants, such as in dependently typed red-black trees (Section 8.4 in [6]).

3 SAT/SMT Integration

There are currently two approaches to SAT/SMT integration into formal proofs: the autarkic and sceptical approach [3].

In the skeptical approach we require the external solver to generate a *trace*, i.e. the steps required to reconstruct a result. This can be used to reconstruct the proof object inside the theorem prover (thus making sure our proofs stay constructive; classical logic is incompatible with some axioms as mentioned in 2.2 and [6]) and to *a-posteriori* verify the SAT/SMT solver. This approach combines the speed of external solvers while reducing the impact of possible bugs.

Finally the autarkic approach embeds a solver inside the theorem prover, verifies its correctness and then uses it for computations. This approach ensures that the solver is total since it's written in a total language and requires least amount of glue code, but doesn't necessarily have performance comparable to well-optimized solvers like ZChaff or VeriT.

3.1 The skeptical approach

The skeptical approach works by using an (external) oracle `oracle : input -> certificate` and then verifying the certificate using a Coq function `checker : input -> certificate -> output`.

For example, a SAT solver integration would have `input` be a CNF formula, `certificate` be a resolution chain or variable assignment and `output` be a boolean that's true when the certificate is valid (e.g. $x \wedge \neg x$ is UNSAT, so calling the checker with a resolution chain $\{x\}.\{\neg x\} \Rightarrow \square$ returns true and calling it with an assignment $x = 1$ returns false).

Here, correctness is simply verifying that for all formulae and certificates, if the checker returns true then the formula has a solution. Unfortunately, bugs in the solver can cause not all formulae to be solved. Furthermore, the solver doesn't necessarily have to terminate.

Given that we delegate the certificate generation to an external solver, this procedure is suitable to problems where certificate generation is hard, but verification is easy; a classic example are NP-complete problems. Difficulties here arise from finding a common format between Coq and the external solver – proof complexity is extremely dependent on having an appropriate representation and Coq's preference of constructive logic also might require an unusual representation (such as when on pen-and-paper, theorems are often proven by contradiction; in constructive logic this only proves $\neg\neg P$).

This approach is used for a SMT-Coq integration in [1, 2]. Use cases include verifying SAT/SMT solver results and using external solvers in Coq tactics. By including a certificate checking step in automated tools (possibly behind a configuration option for speed purposes), reliability of the results of these tools can be improved.

One of the more challenging aspects of that integration was constructing the transformation from Coq to SAT/SMT solver inputs and from resulting solver proof traces to Coq proof objects in an efficient way. The computational power available in Coq is rather limited, and needing too much time to solve a theorem can have negative impacts on usability. However, as you can see in the benchmarks below, SMTCoq is rather fast compared to other solutions.

That integration can also be used for mathematical proofs, as seen in [11], a proof of the odd-order theorem. A subproblem in that proof was solved by encoding it as a SMT formula. The automated proof is 'shorter than our initial version, compiles twice as fast, and is intellectually more satisfying, as it eliminates unnecessary steps from the original proof'. More examples can be seen by following the citation graph for [2].

3.2 The autarkic approach

In the autarkic approach the solver itself is implemented in the formal system. For Coq that means you have a function `decision_procedure : input -> output`, skipping the certificate part. Then the decision procedure itself is proven terminating and correct.

Obviously the guarantees given by doing that are significantly stronger. A program invoking an uncertified solver is necessarily partial. The external solver might fail to terminate or return an incorrect result (which is going to be caught by the checker, but still causes a runtime error). On the other hand, termination proofs can be anywhere from tedious to practically unfeasible or theoretically impossible and proving full correctness of a solver requires that every efficiency-improving "trick" has to be proven correct. For example in [5], a 9 line paper proof required an over 700 line formal proof using Isabelle. Part of the reason is that in a paper proof, many details are often omitted or glossed over — inductive reasoning is hidden behind ellipses like in $(K_1, ..., K_i, ..., K_n)$, existence proofs are missing, and so on. For a high-level human-readable proof, that is often fine and even appreciated. Unfortunately, it doesn't provide the strong guarantees a formal proof does. While a human reader can fill in all these details, computer-based automation techniques currently do not have the capability to handle induction well.

Reflexive SAT integrations such as [14] usually restrict themselves to simple DPLL procedures and the existing SMT solvers such as `omega` for quantifier-free Presburger arithmetic are rather slow.

In the SMT solver Alt-Ergo, the core of the solver was proven correct [13] using Coq. It is then integrated into Coq using the Ergo module. This combines the performance benefits of the skeptical approach with the totality and correctness guarantees of the autarkic approach. Unfortunately, there are still orders of magnitude in performance differences between the skeptical approach used in [2] and the Ergo-Coq approach. The following table from [2] compares the Ergo tactics `dplln` for SAT and `cc` for Equalities + Uninterpreted Functions (EUF) with the SMTCoq [1] tactics `zchaff` for SAT via ZChaff and `verit` for EUF via VeriT:

Table 1 Comparison between Ergo-Coq and SMTCoq

SAT	`dplln`	`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

All times are in seconds. The formulae used for testing are

- for SAT, the de Brujin formulae $deb_n = \forall x_0, ..., x_{2n} : \ (x_{2n} \leftrightarrow x_0) \vee \bigvee_{i=0}^{2n-1} (x_i \leftrightarrow x_{i+1})$
- for EUF, the formulae $D_n = \forall f : (\bigwedge_{i=0}^{n-1} (x_i = y_i \wedge y_i = f(x_{i+1})) \vee (x_i = z_i \wedge z_i = f(x_{i+1}))) \rightarrow x_0 = f^n(x_n)$

4 Conclusion

We provided an introductory tutorial to Coq, giving an overview of the underlying type theory, usage of inductive and dependent types, and encoding logical predicates. Finally we presented both skeptical and autarkic approaches to SAT/SMT integration, including benchmarks between SMTCoq and Ergo-Coq. So far, skeptical solvers appear to be orders of magnitude faster, and the effort required to interface Coq to them is rather manageable. Furthermore, most recent developments in Coq abandoned the idea of fully autarkic solvers, opting for either a proven core (Ergo-Coq) or a-posteriori verification (SMTCoq). A major difficulty in proof automation is the rather unsatisfactory support for induction. Improving this would be a big step towards improving the transition from paper proofs to formal proofs.

Related works

To learn more about Coq, check the Coq manual [7], CPDT [6] or Coq'Art [4]. A more interactive course is Benjamin C. Pierce's Software Foundations, available at [16] For Coq usage in mathematics, see the Homotopy Type Theory book [17] and Univalent Mathematics [18].

References

[1] SMTCoq. `https://github.com/smtcoq/smtcoq`.

[2] 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.

[3] H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.

[4] 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.

[5] J. C. Blanchette, M. Fleury, and C. Weidenbach. *A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality*, pages 25–44. Springer International Publishing, Cham, 2016.

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

[7] Coq Development Team. The coq proof assistant reference manual, 2016.

[8] T. Coquand and G. Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[9] T. Coquand and C. Paulin. *COLOG-88: International Conference on Computer Logic Tallinn, USSR, December 12–16, 1988 Proceedings*, chapter Inductively defined types, pages 50–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[10] G. Gonthier. A computer-checked proof of the four colour theorem. 2008.

[11] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ITP'13, pages 163–179, Berlin, Heidelberg, 2013. Springer-Verlag.

[12] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[13] S. Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq.* Thèse de doctorat, Université Paris-Sud, Jan. 2011.

[14] S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *In Proceedings of TPHOLs*, 2008.

[15] S. Lescuyer and S. Conchon. *Frontiers of Combining Systems: 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, chapter Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme, pages 287–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

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

[17] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[18] V. Voevodsky, B. Ahrens, D. Grayson, et al. *UniMath*: Univalent Mathematics. Available at `https://github.com/UniMath`.